Fall 2018

SCALABLE WEB SERVICE DEVELOPMENT WITH AMAZON WEB SERVICES

Patrick Russell McElhiney
University of New Hampshire, Durham

SCALABLE WEB SERVICE DEVELOPMENT WITH AMAZON WEB SERVICES

BY

PATRICK R. McELHINEY

B.S. Marketing Management, Daniel Webster College, 2010

THESIS

Submitted to the University of New Hampshire

in Partial Fulfillment of

the Requirements for the Degree of

Master of Science

in

Information Technology

September, 2018

This thesis has been examined and approved in partial fulfillment of the requirements for the degree of M.S. in Information Technology by:

Thesis Director, Dr. Michael Jonas, Assistant Professor in Computing Technology

Dr. Mihaela Sabin, Associate Professor in Computing Technology

Timothy Chadwick, Associate Professor in Computing Technology

On August 9th, 2018

Original approval signatures are on file with the University of New Hampshire Graduate School.

ACKNOWLEDGEMENTS

I would like to thank my family, friends, instructors, colleagues, and customers that have all contributed to helping me develop my skills and my marketing & IT firm, MCE123, which prepared me for such a large project for this thesis over the past 20 years. Specifically, I'd like to thank my mother, Diane J. Labrie, for all her dedicated support over the years and throughout my master's degree program. I'd also like to thank the professors that are on the committee to approve this work, including Dr. Jonas, Dr. Sabin, and Professor Chadwick, who have all helped me develop my IT skills and knowledge over the past two years. I would also like to thank all my doctors who made it possible for me to overcome my multiple chronic medical conditions that would have interfered with my ability to produce this document and earn this degree.

To Dr. Jonas, I really appreciate the extra time that you have spent with me on projects, including coming up with the ideas for MeAndYou. I appreciate how much time you have invested into developing my academic life, which has inspired me to continue my education in the Computer Science PhD program at the University of New Hampshire (UNH).

To Dr. Sabin, I have become a better computer programmer because of your rigorous expectations and the structure of the coursework. I appreciate the kind words you have conveyed to my family in support of my continuing education in Durham.

To Professor Chadwick, I am always so impressed with how much you know, and I only wish I had more time to spend learning more from your gigantic brain.

TABLE OF CONTENTS

ABSTRACT

SCALABLE WEB SERVICE DEVELOPMENT WITH AMAZON WEB SERVICES

by

Patrick R. McElhiney

University of New Hampshire, September 2018

The objective of this thesis was to explore the topic of scalable web development, and it answered the question, “How do you scale a website to handle more traffic at peak times without wasting resources?” This is important research to any web company that has issues with rising costs as demand for their website increases. It would be wise for every online business to be prepared for more web traffic, before it occurs, without spending the budget of a multi-million user web company in low traffic periods. The last thing you want is an error as your customer base starts to arrive, giving them a bad experience for their first impressions, which would result in lost revenue.

Scalable software development architectures, including microservices, big data, and Kubernetes were studied, in addition to similar web service companies including Facebook, Twitter, and Match.com. A scalable architecture was designed for a social media web service, MeAndYou, using the big data configuration with a shared Aurora database, which was configured using an auto-scaling group attached to a load balancer in Amazon Web Services (AWS). It was tested using a custom threaded Selenium-based Python script that applied simulated user load to the servers. As the load was applied, AWS added more Elastic Compute

Cloud (EC2) instances running a virtual disk image of the web server. After the load was removed, the instances were terminated automatically by AWS to save costs.

Countless steps were taken to make the web service bigger and more scalable than it originally was, before testing, including adding more fields to user profiles, adding more search types, and separating the layers of code into different Hypertext Preprocessor (PHP) files in the front-end. A version control system was configured on the servers using GitHub and rsync. The systems architecture designed suggests the Match Engine should use a stream processing message queue, which would allow the system to factor searches one at a time as they are created, with horizontal scaling capabilities, rather than grabbing the entire database and storing it in memory. The backend Match Engine was also tested for accuracy using Structured Query Language (SQL) injection, which determined how the match algorithm should be improved in the future.

CHAPTER 1

INTRODUCTION

1.1 Background

Scalability of web service development means that the web hosting infrastructure grows automatically as more visitors access the website, and that the web hosting infrastructure terminates unneeded instances of web servers when they are no longer needed. Websites generally have different traffic patterns caused by variations in the target market at different times of the day – as well as on different days, weeks, months, and years. What this thesis will answer, is how do you scale a web service infrastructure architecture to handle more traffic at peak times without wasting resources.

The U.S. Federal Government had a serious problem with scaling out healthcare.gov. The original budget for the website, which only served around 12 million people, was $93.7 million, but the cost ballooned to $1.7 billion because of poor planning. It's the grim reminder of how money can be wasted if you don't use the right architecture [1].

This research will be working with Infrastructure as a Service (IaaS) provider Amazon Web Services (AWS), which offers cutting edge of cloud computing services that meet the needs of this project. However, restricting a solution to one service provider, such as AWS, is not an all-around win-win situation for all scalable web service design challenges. Specific brand names, specific programming languages, specific software, or specific brands of hardware are not what

solve the problems demonstrated through scalable web service design. Only the systems architecture can solve the scalability problem.

Information Technology (IT) services should be chosen based on their performance and cost, rather than their marketing appearance. An implementation of a system is simply a snapshot of the architecture. The best architectures are not celebrated through bragging about their open source or commercial implementations, like “Ubuntu Linux” with “PostgreSQL” and “Apache” out of the box, but instead through vendor and technology agnostic designs.

IT customers should select the web provider that offers the best service for your architecture type. Know that technology-agnostic designs and architectures affect a website’s availability [2]. Providers use Service Level Agreements (SLA), which are commitments with the client that uses the services. An example of an SLA would be “three nines”, or services that are up 99.9% of the time [3]. Even though this looks and sounds like a good deal, it would still allow for up to 8 hours, 45 minutes, and 57 seconds of downtime per year [4]. The higher percentage of uptime, the costlier the SLA becomes. If the online web service is making millions of dollars an hour, having the most possible uptime is ultra-important.

This research used a web service called “MeAndYou”, first developed and prototyped at the University of New Hampshire (UNH) in Manchester. It used a Model-View-Controller (MVC) architecture. Students in computing programs at UNH Manchester have spent thousands of hours working with the project, between designing, coding, deploying, and testing, and there are still a lot of improvements that can be made. In this thesis project, I designed a scalable infrastructure and put it to the test with simulated traffic to determine how the site architecture

would work with hundreds of millions of users accessing the site. This thesis explains the concepts behind the infrastructure choices that have been made, in addition to changes that made the site more scalable.

This research explains the various layers of code in the website's front-end, or Graphical User Interface (GUI), as well as how the database model was constructed, in addition to how a match engine works using an algorithm that make matches between searches from sets of two different users of the web service. It also discusses the need for redundancy in the design – such as using an Amazon AWS Elastic Compute Cloud (EC2) Auto-Scaling Group attached to a load balancer for the front-end. There need to be clear lines between production and development environments. We will explore how the web service can be implemented using various scalable architecture models, including big data and microservices, and an example is using containers deployed on Kubernetes.

1.2 Scalable Software Development Architecture Types

There are two main software development architecture types that apply to developing scalable web services: microservices, and big data. An example of a scalable architecture is a distributed container system called Kubernetes, which is used by AWS [5].

Microservices are developed with little communication between the various teams that work on individual services, which encourages more frequently updates than other cloud development architectures. Microservices require a mature development and DevOps culture, and they have a lot to do with modern day scalable web services. If done right, this programming method can lead to higher release velocity, faster innovation, and resiliency [6]. Microservices

are commonly developed as backend processes and can be written in various programming languages that generally communicate with other services using the Hyper-Text Transfer Protocol (HTTP), or sometimes other asynchronous communication. Microservices is a variant of the Service-Oriented Architecture (SOA), a software development technique that makes a computer application from a collection of loosely-coupled services. One benefit to this type of software development is the ability for teams to enable continuous delivery and deployment in an agile environment [7].

The big data architecture can serve billions of web service users per day, and is based on the concept of the 3Vs. According to Doug Laney, the father of the concept of 3Vs, and others in the industry have suggested that Big Data has four defining properties or dimensions [9]:

- **Volume** – the amount of data.
- **Variety** – the number of different types of data.
- **Velocity** – the speed of data processing.
- **Other** – which may include **Variability** – the increase in the range of values of a large data set, and **Value** – addresses the need for valuation of enterprise data .

The big data software development architecture is designed to handle the input, processing, and analysis of data that is too large or too complex for traditional data storage methods. They typically include one or more of the following types of data workloads:

1. Batch processing of big data sources at rest.
2. Real-time stream processing of big data sources in motion.

3. Interactive exploration of big data.

4. Predictive analytics and machine learning [10].

Distributed systems are comprised of multiple different applications running on different machines, or many replicas of one application running across different machines, all communicating together to implement a complex system such as a web-search engine.

It would also be possible to use a distributed container system such as Kubernetes, and container deployment software like Docker to run instances of the MeAndYou Match Engine. In general, the goal of a container is to establish boundaries around specific resources, such as that *'this specific application requires 2 CPU cores and 8GB of memory'* to run properly. The boundary that the container establishes delineates team ownership, such as that *'a specific team owns a specific image'*, and it is intended to provide separation of concerns, such as that *'this specific image does one specific thing'* [14].

1.3 Objectives

The objectives of this thesis are to study and discuss research conducted about cloud computing services, as well as developing, implementing, and testing a web service architecture to make it scalable with Amazon AWS. This research should leave the reader knowledgeable enough to discuss the topic of scalable web service development with an IT professional that has expertise about the topic, and be able to convey system requirements for making their website scalable in the cloud. The implementation of the developed architecture will be as advanced as can be achieved within the limited timeframe of this research project. The system will be testing

by applying load to the load balancer, and by testing the accuracy of the matching engine at the backend.

1.4 Approach

This research takes the approach of developing a hybrid cloud-based architecture for a web service that takes advantage of the microservices and big data software architecture types. For the purpose of scaling the web service MeAndYou, we utilized Kubernetes distributed system architecture using Amazon AWS. The design, implementation, and testing of the web service offer insights into approaches, tools, and practices that support software architecture scalability.

1.5 Organization

In Chapter 2, this research paper will discuss the history of scalable web service development, including the scalable web service development application examples of Facebook, Match.com and subsidiaries, and Twitter. In Chapter 3, this paper discusses the MeAndYou system overview, and the system architecture that will be designed in a prototype as part of this research. The system architecture is split up into sections based on the front-end, version control system, an AWS Aurora database, a match engine system, and Route 53 Domain Name System (DNS). In chapter 4, this paper will explain how that prototype was developed using the architecture discussed in Chapter 3. Finally, in Chapter 5, this paper discusses the results of this research, including the graphical user interface and front-end testing, in addition to the match engine accuracy testing, and concludes this research paper in a summary of what was done and what can be concluded.

CHAPTER 2

HISTORY OF SCALABLE WEB SERVICE DEVELOPMENT

2.1 Scalable Web Service Architecture Types and Tools

2.1.1 Microservices

Different microservices are separated by their purpose, such as front-end, billing, and logistics. A change to a small part of the application only requires rebuilding and redeploying a small number of services. It's essentially similar to the Internet meeting the UNIX environment, where everything uses pipelines with extremely loose coupling [7]. A microservice architecture consists of a small collection of small, autonomous services, according to Microsoft. Each service is self-contained, consisting of a separate codebase, and should implement only a single capability, which is managed by a small development team. The various services within a microservice architecture don't need to share the same technology stack, common libraries, or frameworks. A diagram depicting the microservices architecture is in **Figure 1** on Page 8 [8].

In addition to the microservices, other components in the application include:

- **Management** – responsible for placing services on nodes, identifying failures, and rebalancing services across nodes.
- **Service Discovery** – maintains list of services and which nodes they are located on, to enable service lookup to find the endpoint for a service.
- **Application Programming Interface (API) Gateway** – receives responses from many services and return an aggregated response. The clients connect to this one

endpoint, which then connects them to the backend with the appropriate service based on their request.

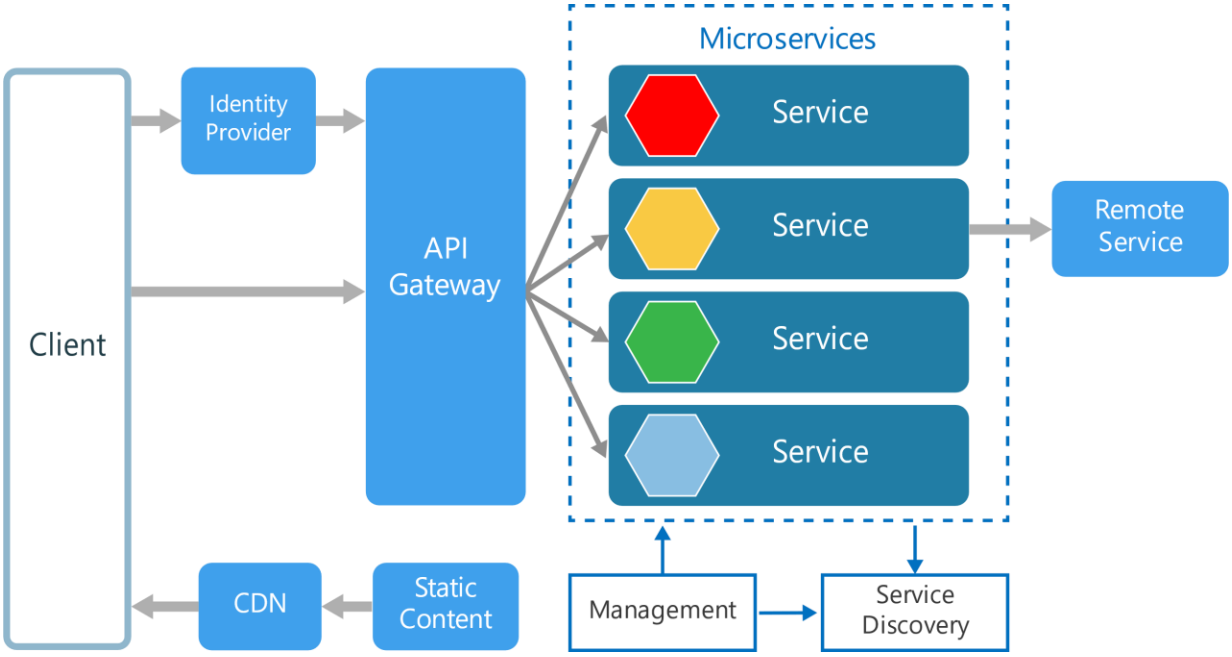

Figure 1: Microservices Architecture
Source: Microsoft Azure: Microservices architecture style, Page 1 [8]

This architecture benefits large, complex applications that need to be highly scalable, and require a high release velocity, based on rich domains with many subdomains in organizations that consist of lots of small development teams. The advantages of a microservices-based architecture include independent development and deployments from small, focused teams, fault isolation, mixed technology stacks, and granular scaling. This allows the application to be scaled based on the individual Virtual Machine (VM) – whether it be a high or low CPU load, with high or low memory, and with high or low disk space, among other scalable attributes [6].

Microsoft explains microservices well, which is discussed in greater detail over the next several pages below. Designing software with this architecture requires the individual

development teams to be decentralized from each other – not sharing code or data schemas with each other. The storage used for each service may vary, depending upon what the best type of storage is for that service. The services should never share storage. The services should communicate with each other through well-designed APIs that use encapsulation, or in other words they don't leak their implementation details with other services or clients. The APIs should be based on the domain, not the internal implementation of the specific service for which it is designed for. There shouldn't be any coupling between services, such as shared database schemas or rigid communication protocols. The concept of the API is essential to microservices.

Certain services such as authentication and Secure Sockets Layer (SSL) termination should be done by the gateway – not by the services. The attributes of the application domain should not be shared with the gateway. The gateway should be able to route requests from clients to services without knowing internal domain knowledge that applies to the services – or in other words, the gateway should be de-coupled from the services.

The services should be loosely coupled, and they should have high functional cohesion. Functions that are likely to be changed at the same time should be packaged together and they should also be deployed together within one service – otherwise the different services that have the various functions that rely on each other would become tightly coupled, since a change to one of the services would require a change to the other service(s) as well. When two services are chatty together, this is a clear indication that they are tightly coupled with low cohesion. The services should isolate failures, to provide resiliency, and to prevent the failure of one service from cascading to another service.

Microservices communicate using asynchronous messaging, like in Object-Oriented Programming (OOP), however it is nearly the equivalence of decentralizing each class into a separate service, while a service may consist of multiple classes and isn't a class its self. Each individual microservice has everything it needs to provide its services to the API gateway, internal to the service. Microservices can be scaled horizontally, adding new instances as the load increases. Examples of microservices are the GUI of the website, a logging system, and a communication system that sends e-mails based on events such as new data elements being added or modified in the database. A database application could be split up into multiple microservices, such as separate services for reading data, verifying data, updating data, and deleting data, as well as a scripted trigger system that makes a Remote Procedure Call (RPC) every time a certain Jane Doe's record is updated.

The application state is distributed, and as such operations are done asynchronously and in parallel of each other. The fact that microservices can run of different servers means that they can have dedicated Central Processing Unit (CPU) cores, and for the most part they won't have to wait for computing of other processes to complete. Deployments of microservices are automated and predictable in their runtime activities. In the modern cloud, services are designed for elastic scaling with polyglot persistence, or the diversity of storage technologies, accomplishing eventual consistency – comparatively to the strong consistency of traditional on-site deployments. Services are designed for failure with a Mean Time to Repair (MTTR) based on the SLA between the cloud platform and the company. The system in the cloud, overall, is designed for small and frequent updates because of the decoupled nature of the microservices. The infrastructure used for microservices is considered immutable, or in other words it can't be

modified. The application should favor a Platform as a Service (PaaS), like those offered from Microsoft AZURE, Amazon AWS, Google Cloud, or other popular hosting companies like Digital Ocean.

Microsoft suggests the following top ten design principles to make the application more scalable, resilient, and manageable:

- **Design for Self-Healing** – to design the application to correct errors that arise at runtime.
- **Make All Things Redundant** – avoid having a single point of failure by designing the application with redundancy.
- **Minimize Coordination** – make the microservices loosely coupled to allow for scalability.
- **Design to Scale Out** – allow for horizontal scaling, so that new instances can spin up when needed, and be terminated when they are no longer required.
- **Partition Around Limits** – use partitioning to bypass compute, database, and network limits.
- **Design for Operations** – make sure the operations team has what they need in the application.
- **Use Managed Services** – Platform as a Service (PaaS) is preferred over Infrastructure as a Service (IaaS).
- **Use the Best Data Store for the Job** – pick the storage technology that best fits the data and how the data will be used, for each microservice.

- **Design for Evolution** – allow for change that will occur over time, as part of the continuous innovation process.
- **Build for the Needs of the Business** – the design requirements should reflect the requirements of the business.

There are challenges when building a microservices-driven application, including its complexity and lack of governance. The entire system is more complex than a monolithic application – it may end up using more framework and languages, making it difficult to manage. Data only reaches eventual consistency since each service has its own database. One common oversight is not having backwards and forwards compatibility with the other services – which could lead to downtime of the application as upgrades occur. The application could become chattier in-between the various services, which will increase congestion and latency of the network – and therefore it is important to make sure that the services are decoupled. There's also the issue of correlated logging, that is required to keep track of a user's activity by different services – which is important to have for debugging and for making improvements to the architecture [6].

2.1.2 Big Data

Microsoft explains the big data software development architecture very well, which will now be discussed over the next several pages below. They describe that the big data architecture should be used to store and process data in volumes too large for a traditional database. It should also be used to transform unstructured data for analysis and reporting. Additionally, it can be

used to capture, process, and analyze unbound streams of data in real time with low latency. A diagram depicting the big data architecture can be seen below, in **Figure 2**.

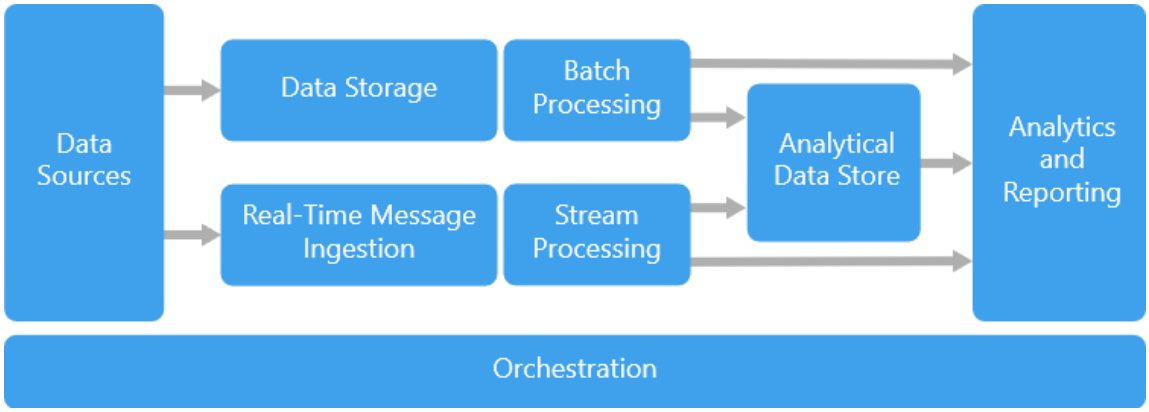

Figure 2: Big Data Architecture
Source: Microsoft Azure, Big Data Architecture Style, Page 10 [10]

The benefits of using the big data architecture include a wide variety of technology choices, including Amazon AWS, Microsoft AZURE, and the Google Cloud Platform (GCP). The big data architecture takes advantage of parallel processing, which can process large amounts of data with little latency. The big data architecture also supports scaling out, so solutions can be customized to large or small sizes. It also works well with the Internet of Things (IoT), and Business Intelligence (BI) systems such as Customer Relationship Management (CRM).

The big data architecture has been used, most notably, with social media services such as Facebook, Twitter, and LinkedIn. It allows large volumes of data to be processed and leveraged for application purposes, that can scale to billions of active users daily.

Most big data architectures include some or all the following components:

- **Data Sources** – they start out including relational databases, document databases, web logs, or application logs.
- **Data Storage** – data is typically stored in a dynamic distributed file store that can hold a wide array of data sources of varying content types, such as images, text, 3D models, etc.
- **Batch Processing** – this is used to compute batch jobs from the data, such as to filter, aggregate, or otherwise prepare it for use in data analytics.
- **Real-Time Message Ingestion** – if the data includes real-time sources, the system must use stream processing to have a buffer for messages, and to support scale-out processing, reliable delivery, and other types of message queuing services.
- **Stream Processing** – this is to support capturing data messages in real time, and preparing them for data analysis.
- **Analytical Data Store** – this involves storing the data in a system that can be queried, such as a NoSQL database.
- **Analysis and Reporting** – this provides insight into the data through analytics and processing.
- **Orchestration** – this can include encapsulating information and moving information in-between different tools in the big data warehouse [10].

2.1.3 Kubernetes

Kubernetes, or K8s, is an open-source container-orchestration system that is used for automating the deployment, the scaling, and the management operations of application containers across clusters of hosts. It was originally developed by Google, and works with a wide range of container tools, including Docker. Kubernetes is being used in several third-party vendor solutions, including Azure Kubernetes Service (AKS) from Microsoft, Elastic Container Service for Kubernetes (EKS) from Amazon, and the Google Kubernetes Engine (GKE). A diagram depicting the Kubernetes architecture can be seen below, in **Figure 3**.

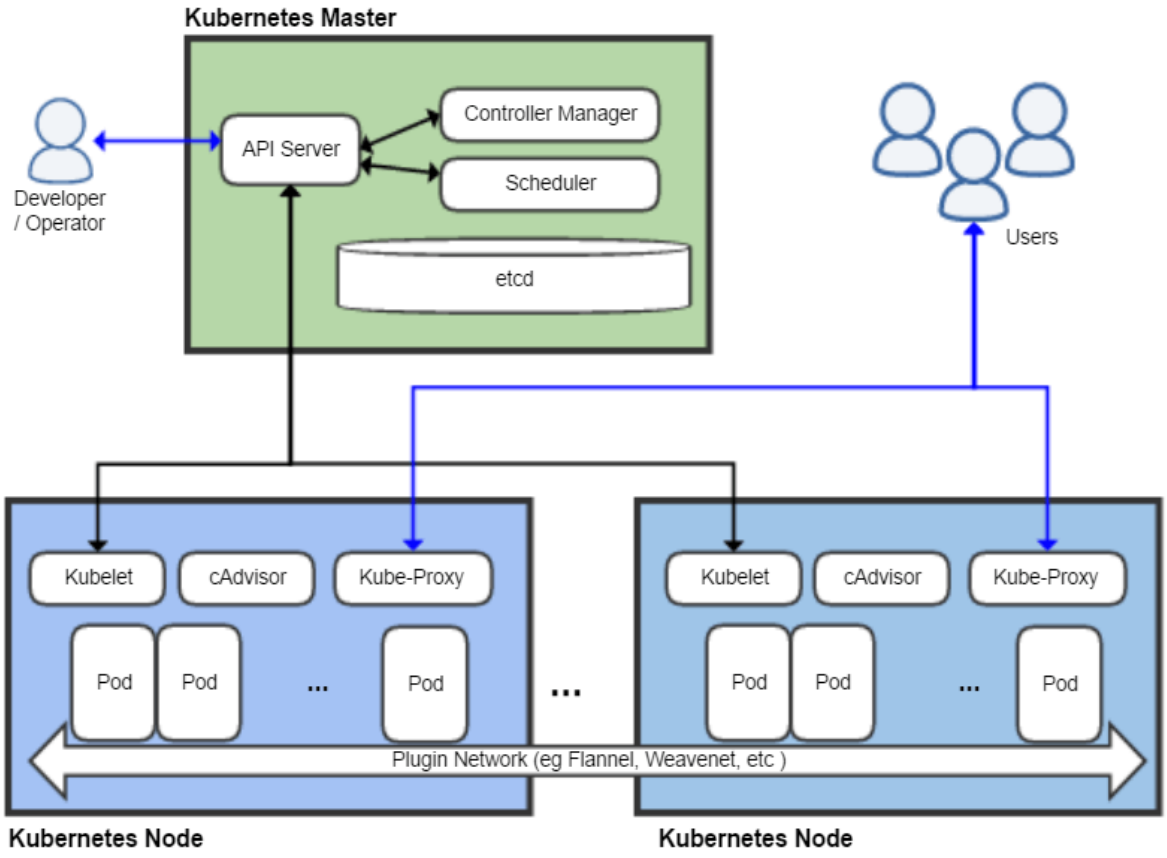

Figure 3: Kubernetes Architecture Design
Source: Khtan66 on Wikipedia.org - Own work, CC BY-SA 4.0 [11]

Kubernetes was designed based on a decade of experience deploying reliable, scalable distributed systems in containers through application-oriented APIs at Google. As more and more services are offered over the Internet using APIs, these distributed systems must be highly reliable. Even if a portion of their system crashes or fails, the system must not fail, and they must be available always – even during software upgrade rollouts and maintenance events. These systems must also be highly scalable because more and more businesses are coming online using the services.

Kubernetes defines a set of building blocks known as "primitives", which collectively provide mechanisms for deploying, maintaining, and scaling applications. The components that make up Kubernetes are:

- **Pods** – the basic scheduling unit in Kubernetes, that adds a higher level of abstraction by grouping containerized components together. A pod contains one or more containers. Each pod in is assigned a unique Internet Protocol (IP) address within the cluster. A pod can also define a disk volume, such as a local disk directory or a network disk and expose it to any containers that are inside the pod. A pod can be managed manually through the Kubernetes API, or the management of specific pods can be delegated to a controller.
- **Labels and Selectors** – key-value pairs attached by users to any API object in the system, such as pods or nodes. Selectors are queries against labels that resolve to matching objects. Both labels and selectors are the primary grouping mechanism in Kubernetes, used to determine which components apply to an operation.

- **Controllers** – a reconciliation loop that drives a cluster's actual state to a desired state, by managing sets of pods. The set of pods that a controller manages is set by label selectors, which are part of the definition of the controller. Types of controllers include:
    - **Replication Controller** – handles replication and scaling of a pods across the cluster. It also handles the creation of replacement pods if the underlying node has failed.
    - **DaemonSet Controller** – used for running one pod on every machine, or a subset of machines.
    - **Job Controller** – used for running pods that run to completion, such as pods that are part of a batch job.
- **Services** – a set of pods that work together, such as single tier of a multi-tier application. The set of pods that constitute a service are defined by their label selectors. By default, a service is exposed inside a cluster, but a service can also be exposed outside a cluster.

Kubernetes was designed to be loosely coupled and extensible, for meeting a variety of different workloads. This extensibility is provided in large part by the API, which is used by internal components, extensions, and containers running on Kubernetes. Kubernetes also provides service discovery and request routing by assigning a stable IP address and DNS name to the service. It can also load balance traffic in a round-robin pattern to an IP address among the pods matching the selector.

There are four benefits of using containers and container APIs:

1. **Velocity** – the speed at which you can develop, including iterative improvements to services every few hours. It's measured in the sense of how many improvements you can make while keeping the system highly available.
2. **Scaling** – relies on decoupled architecture, or that each component relies on APIs for communication, as well as service load balancers. Since containers are immutable, scaling is simple to implement – usually requiring the change of a configuration file and asserting the new declarative state to the container API.
3. **Abstracting Your Infrastructure** – this involves not getting sucked in to any specific cloud provider's infrastructure and not being able to duplicate the application environment anywhere else. In this sense, this project's AWS deployment is simply the first step towards developing a cloud infrastructure – but ideally in the future, the application should be developed to work with a container API, such as Kubernetes, so it can be deployed on any infrastructure.
4. **Efficiency** – there is a concrete economic benefit to the abstraction with a container API like Kubernetes. It can automate the distribution of applications across a cluster of machines, ensuring higher levels of hardware utilization. Time is money, and server resources that aren't being used are a loss of revenue [12].

It's possible to assign different resource requirements and priorities to different containers. For an example, it's possible to use a sidecar pattern to route HTTP traffic from the localhost of one container through the HTTP Secure (HTTPS) server running on the sidecar container. It's also possible to use a sidecar container to host the service, for which there is an API that configures a web server on another container, so that you don't have to login to each of

hundreds or even thousands of web servers to make one change to the configuration on each and every one of them [14].

A container's interaction with the outside world is through the parameters of its API. This makes them modular and re-usable. For an example, a container platform could be built around the git workflow, so that when a git repository is updated, the updates are automatically sent to the servers [10]. Reliability, scalability, and separation of concerns are all properties of real-world scalable systems that are built from many different components, spread across lots of different machines, which are comprised of multi-node distributed patterns that are much more loosely coupled than the software that runs on a single node [14].

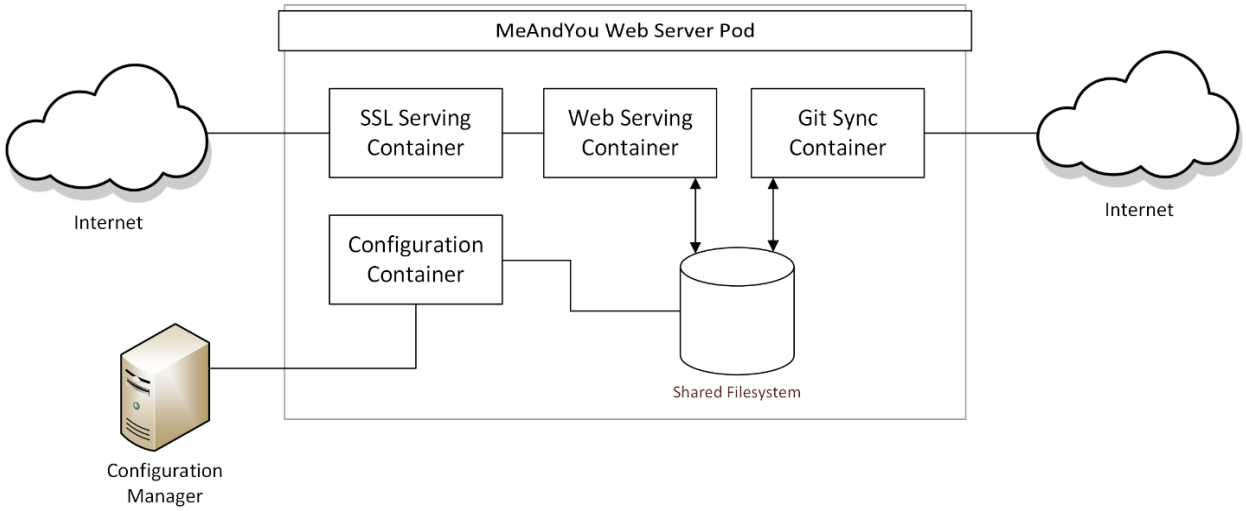

Figure 4: Diagram of Pod Design for MeAndYou's Web Servers

Above, in **Figure 4**, is a diagram of a pod design that was created for MeAndYou's web servers. It would include a sidecar container that synchronized the web server's filesystem with GitHub, and it would have a sidecar container that routes the HTTP traffic through HTTPS, and it

would also have a configuration container that would allow the pod to be monitored and configured remotely, including php.ini and other web server configuration files not in GitHub.

2.2 Scalable Web Service Companies

2.2.1 Facebook

Facebook, the most popular social media network in the world, had 2.19 billion monthly active users as of the first quarter of 2018, as seen in **Figure 5**, according to Statista, whereas "active users" are those that have logged in to Facebook during the last 30 days [15]. The top three nations are India, with 270 million users, the United States (US), with 240 million users, and Indonesia, with 140 million users [16]. Over 1.4 billion users log onto Facebook daily, and are considered daily active users [17]. With all these active users, you might imagine that Facebook has major scalability issues with its website.

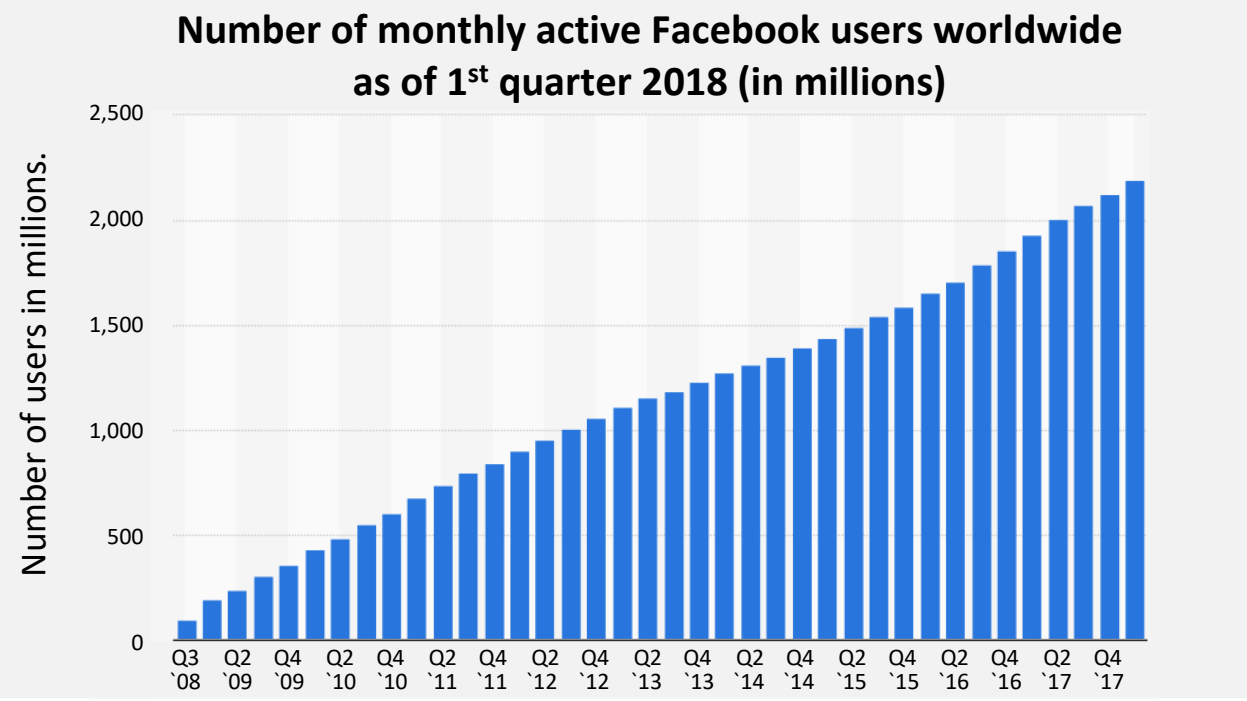

Figure 5: Number of Monthly Active Facebook Users Worldwide as of 1st Quarter 2018
Source: Statista, The Statistics Portal: Number of monthly active Facebook users, Page 1 [15]

Facebook uses a “social graph” HTTP-based API, to allow its web service and third-party applications to interact with the Facebook database, which is a graph database – one that uses nodes, edges, and fields to organize their data:

- Nodes are the individual objects, like a User, a Page, a Photo, or a Comment.
- Edges are connections between a group of objects and a single object, such as Photos on a Page, or Comments on a Photo.
- Fields are data about an object, such as a Page’s name, or a User’s birthday [18].

Facebook’s API allows the web service’s users to upload more than 300 million photographs a day. In fact, in just one minute, an average of 510,000 comments are posted, 293,000 statuses are updated, and 136,000 photos are uploaded. One in every five website pages accessed in the US on the Internet are of Facebook [17]. To handle all that traffic, Facebook’s website uses its own flavor of PHP called HipHop PHP, which was designed by Facebook to speed up operations, improve efficiency, and decrease CPU utilization.

In 2010, Facebook was running thousands of MySQL nodes, and its Senior Open Programs Manager, David Recordon, said they didn’t care that it’s a relational database. However, he said that they don’t use MySQL for joins, or complex queries that are pulling multiple tables together in the database. He said that Facebook has three layers of its data infrastructure:

1. **MySQL** – for Primary Data Store
2. **Memcached** – for caching of data
3. **Web Server** – to serve the data

Recordon said that the joins of the data from multiple tables are done by HipHop PHP. Additionally, Facebook used NoSQL-type databases, including more than 150TB stored in Cassandra, and more than 36PB stored in a Hadoop Distributed File System (HDFS). He said most of the user data is stored in MySQL. At the time in 2010, Recordon said that Facebook had 2,200 servers running 23,000 CPU cores [19].

Facebook has done more than just host big data – they've created major open-source tools, including:

- **Cassandra** – a highly scalable, eventually consistent, distributed, structured key-value database storage system. It was designed to handle a wide array of data spread out across many servers, and powers Facebook's inbox search feature.
- **Thrift** – a lightweight remote procedure call framework for scalable cross-language services development, that supports C++, Erlang, Java, Perl, PHP, Python, and Ruby, among other languages. It was created to save on development time, providing a division of work to be done on high-performance servers.
- **Scribe** – a log server for aggregating data streamed in real-time from many servers. Built on top of Thrift, it is a scalable framework used for logging vast information [20].

Facebook ranks as the third busiest website according to Alexa's page rank [21]. Processing one single profile page on Facebook generally includes querying hundreds of servers, that process tens of thousands of individual pieces of data, and then deliver them to the requesting user's computer in less than one second. Facebook launched its own company-built and operated datacenter in Prineville, Oregon, in April 2011, and has since built additional

datacenters in Forest City – North Carolina, Lulea – Sweden, and Altoona – Iowa. Facebook has also broken ground to build additional datacenters in Fort Worth – Texas, Clonee – Ireland, and Los Lunas – New Mexico. The company also leases datacenter space in Ashburn – Virginia, as well as Singapore [22].

Facebook's datacenters in Prineville, including an expansion datacenter, were slated to take up 307,000 square feet of rack space [22]. The company also secured a third location in Prineville for an additional 487,000 square feet of rack space in September of 2015. This compares to only 144,000 square feet for an entire Costco shopping center [23]. Walking around a Costco can give you quite a work out – they even have their own restaurant that sells full size pizzas and hot dogs, in addition to ice cream. The size comparison is utterly unthinkable. Even bigger than Wal-Mart. Scalable web services take up a lot of resources – including physical real estate.

The demand for these data centers is consistent with the ever-growing demand for Facebook, not only in the US, but around the world. It's a phenomenon. Facebook is more than a social network – Facebook has open source projects for developers, such as Artificial Intelligence Tools, Augmented Reality Studio, Games, Business Tools such as the Marketing API and the Instagram Graph API, in addition to Publishing Tools, Social Integrations like Sharing and Social Plugins, and Virtual Reality [24]. With the ever-growing world of Facebook, there doesn't seem to be any chance that it could be overtaken by a startup concept like MeAndYou.

Facebook stored a total of around 300 PetaBytes (PB) in April of 2014, with an additional 600 TeraBytes (TB) *[average]* of new data coming in every single day. They say that Facebook's

data storage capacity has grown 3x larger than it was a year prior. They were adopting techniques such as “cold storage centers” and Redundant Arrays of Independent Disks (RAID) in HDFS to reduce replication ratios within their drive arrays. They use Record-Columnar File Format (RCFile) in their data warehouse, where they calculate what types of advertisements to display for various users, and other event-driven data analytics. They use the Facebook ORCFile writer, to create sorted dictionaries, which is available on GitHub and is part of the Apache Hive project [25].

Facebook has also developed and deployed the Binary Optimization and Layout Tool (BOLT), which optimizes the placement of instructions in registry inside the CPU, to reduce CPU execution time in half, and it’s also an open-source project. Facebook says the machine code for one of their services can be hundreds of megabytes, which doesn’t fit in a CPU’s registry [26].

2.2.2 Match.com

Match.com, the largest dating website in the world, serving 25 countries in 15 different languages, based out of Dallas, Texas. It went live in 1995 as a beta version website, providing free dating services in the US. The initial users were given free lifetime memberships to market the web service and build its user base. It was recognized by the Guinness Book of World Records in November 2004 as having had more than 42 million singles register, with over 15 million active site users from around the world. The company has acquired several other dating websites, including BlackPeopleMeet.com, SinglesNet.com, Chemistry.com, OurTime.com, OkCupid.com, Tinder, Meetic, Twoo, LoveAndSeek.com, SeniorPeopleMeet.com, and PlentyOfFish.com [27].

Match has made several technological advances over the years, including launching a mobile app called “Stream” in April 2014, like the Tinder app. ComputerWorld defines that the following attributes of a dating site are important to attract users:

- **Offer Excellent Response Times** – because people want instant gratifications, the website must offer connections that respond to messages, quickly.
- **Convert at least 10% of Visitors** – a tenth of users convert to paying customers.
- **Deliver Acceptable Range of Probable Matches** – including different ways to communicate with them, such as video chat and photo-realistic avatars.
- **Keep the Quality of the Prospect Pool High** – including deactivating fake user accounts, scammers, con artists, criminals, and sexual predators.

PlentyOfFish.com, a subsidiary of Match, operates on just three web servers, five messaging servers, and five database servers, with a database with just 200GB of data. This is because it uses a short questionnaire – yet it still serves up 200 billion pages a month to 12 million active users. In retrospect, True.com uses around 200 servers, including a 64-bit 32 processor Unisys server running Microsoft SQL Server. Their database is 1TB in size. eHarmony, another Match competitor that is said to have the most comprehensive matching system, pulls more than 1TB of data a day from its Oracle database into a high-performance data warehouse [28].

Match.com is powered by the Synapse algorithm, which learns about its users in the same way that Amazon recommends products to shoppers. The site uses Chemistry.com to get personalized survey data on users, and their preferences. The algorithm needs to determine that

Person A is interested in Person B, while at the same time Person B is interested in Person A. What also makes things difficult, is that people tend to lie about their own attributes, such as marital status, and their weight. The system needs to be able to figure out what data is real, and what data is fake [29].

2.2.3 Twitter

For Twitter, it isn't just about scalability, but also reliability and efficiency, so the site can handle events like the Super Bowl, the World Cup, and New Year's Eve celebrations. On their online blog about the IT infrastructure, Mazdak Hashemi, Vice President of Infrastructure and Operations, talks about some of the changes that have occurred at Twitter to improve uptime. He says that downtime, in the past, has been due to software limitations – not just hardware problems. This may have had to do with the fact that the software wasn't designed to have hardware problems, such as loss of power, loss of Internet connectivity, or other major hardware failures. However, he admits that some mistakes were made with hardware decisions, such as the decision not to buy a second power supply for their initial set of servers. Even I keep spare power supplies in stock for my small company with one server in-house, and I don't run a large Internet-based company – so I found this astonishing that Twitter tried to cut corners with their hardware.

They eventually learned to model failures into their software, so that outages would be counter-acted to better distribute services across the domain. They added additional data centers in other availability zones, so that if there was a natural disaster in one area, another

datacenter would pick up where the troubled datacenter went down. They have also improved their data centers with regards to power usage, to improve power efficiency.

Twitter has begun testing the efficiency of off-the-shelf hardware, which was recommended in Building Scalable Websites, to reduce hardware costs – so that instead of buying proprietary hardware that makes you dependent upon a specific manufacturer, you spread out your purchases to multiple manufacturers for hardware that is readily available for purchase from major retailers. They've found that, for an example, some RAID controllers interfere with the efficient operation of Solid State Drives (SSDs), and can cost as much as a third of the expense of one server. As they grew, they invested in a team to create white box solutions that focused on reducing costs and increasing performance metrics.

Twitter made their primary storage method SSDs in 2012, and integrated Hadoop in 2013. They developed a key/value storage system in 2014, switching the main storage method to NoSQL databases. Later, in 2016, they integrated Graphics Processor Units (GPUs) for interference and training of Artificial Intelligence (AI) / Machine Learning models.

Twitter has two main goals with its infrastructure management – and that includes removing unused components and improving server utilization. Twitter's workload is split up into four main verticals, which are storage, compute, database, and GPU. They optimize component selection where the systems might go underutilized or unused. They have done things such as improving the thread count of their CPUs, and increasing the power efficiency of threads, to build a more versatile system architecture [30].

CHAPTER 3

DESIGN OF MEANDYOU – A SCALABLE WEB SERVICE

3.1 Approach

MeAndYou is an example web service that was designed with the scalability of both the microservices and big data architectures in mind. It was designed using the AWS cloud hosting platform, which uses Kubernetes to host containers which can serve as databases, virtual machines, and other types of instances. The approach to building this web service was to incorporate as many characteristics of cloud computing scalability as possible, as researched ahead of time. Shown in **Figure 6** below, the web service takes in user requests from a load balancer and redirects it to any number of clients that are available on web servers, managed by an AWS Auto-Scaling Group. The web clients work through an API to access and store data in a central database, and the front-end sends messages to a message queue, which is stream processed to make matches based on different search types.

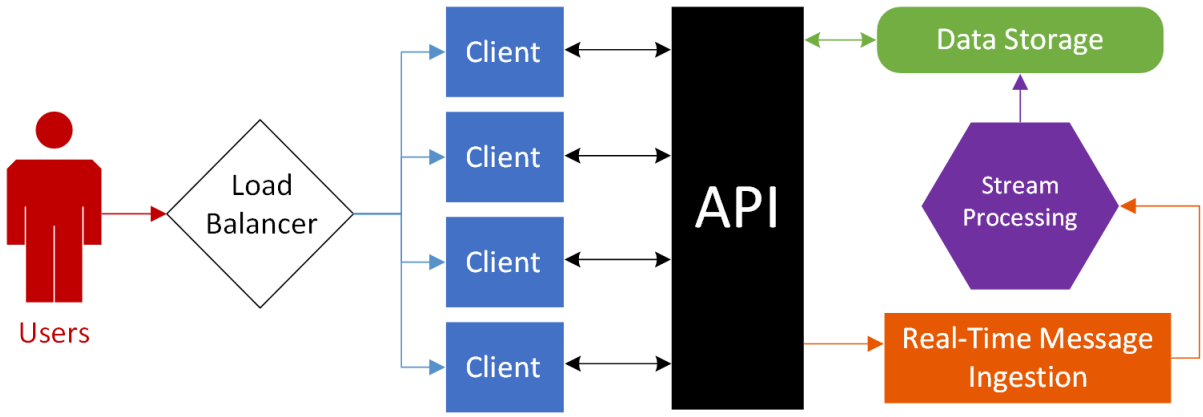

Figure 6: MeAndYou High-Level Hybrid Architecture Diagram

The purpose of this project was to research, develop, implement, and test the MeAndYou web service systems architecture to make it scalable using AWS.

3.2 MeAndYou System Overview

MeAndYou is a black-box context-dependent search submission web service that has a matching engine, which currently uses the Jaro-Winkler distance algorithm. The match engine finds search records that are looking for each other in a central database, with approximate distance calculated up to a threshold. The search data is inserted by users that interact with the web service hosted on a web server. For there to be a match between two users in the MeAndYou web service, they must have been searching for each other, using each other's personal information as is present in each of their user profiles, and the searches must have occurred within a specific window of time, which is determined by each of the five search types – otherwise the searches can expire in different timeframes, and the users would not be matched to each other at all.

When two people who already know each other use the MeAndYou website to search for each other – both using the same search type, with their own private accounts that they have registered for and have entered their personal information into, the matching engine determines if those searches correlate to each other based on the personal data that was saved in the database from each of the users filling out the search form on the website, compared to each of the other users' personal profiles. Additionally, neither the personal profiles nor the searches are public facing. Only the user that has made the search can see the search data, and only the user

that has filled out the private profile can see the profile data. Even when there is a match, neither the profile nor the search data are shared.

Only when a match has been made, the system displays that those two users are matched, and it only displays that they are a match to each other on each of their dashboards – not to the public. Additionally, the system only shares the actual name of the match, and their e-mail address in the form of a link on the match's first and last name, together – and only to the other site member that they have been matched to – never to the public.

MeAndYou doesn't use pictures like popular social media web services like Facebook use, where a user can browse pages with pictures and see who they recognize, and then click to add them as a friend. With MeAndYou, the user must know the person that you're searching for, and they must know the user as well, and they both need to be searching for each other with accurate information that matches an accurate representation of each other with their personal profile within the same period.

There are currently five different types of matches that can be made on MeAndYou:

- **Crushes** – each user can have 6 Crushes, and the searches expire after 14 days.
- **Friends** – each user can have 100 Friends, and the searches expire after 30 days.
- **Family Members** – each user can have 100 Family members, and the searches expire after 30 days.
- **CoWorkers** – each user can have 100 CoWorkers, and the searches expire after 30 days. The user must fill out their Company Name and their Job Title.

- **Long Lost Love** – each user can have one Long Lost Love, and the search expires after one year. This search type requires additional information to be filled out.

3.3 MeAndYou Architecture Overview

The objective for this project is to configure an auto-scaling group of XAMPP-based web servers, or Cross-Platform (**X**), Apache (**A**), MariaDB (**M**), PHP (**P**) and Perl (**P**), that connect to an Amazon AWS Elastic Load Balancer. This will allow the testing of more than one web server hosting the same MeAndYou web service, connected to a central database. Several different infrastructure designs that would handle hundreds of millions of concurrent visitors have been developed, however due to limitations on time and financial resources, they will not be deployed and tested at those scales. It is impractical for one person to develop and deploy a web service for millions of people for this thesis, with limited time.

The design that this project will be based on is much smaller, which includes an EC2 auto-scaling group of web servers, that all automatically update their code from a single EC2 Version Control instance, which will allow changes to be made to the multiple web servers without having to FTP into each of the EC2 instances every time there is a change in code. This design uses an AWS Aurora database instance, with a pre-defined capacity of 1TB. Amazon Aurora was designed to be compatible with MySQL databases, which the original MeAndYou website used. Were the infrastructure to be suitable for hundreds of millions or more users, DynamoDB or another NoSQL database would be used in its place.

This infrastructure design, that will be used to test the theory of scalable web service development with AWS, separates out the web server, the database, and the match engine into

separate VM instances. This is unlike the original model of MeAndYou, in which all three components of the MVC architecture – the Front-End, Database, and Match Engine – were all based on the same server. Connecting to the server instances will rely on the Transmission Control Protocol and Internet Protocol (TCP/IP) stack, such as Secure Shell (SSH), Remote Desktop Protocol (RDP), and File Transfer Protocol (FTP) to make changes to the configurations and code.

3.3.1 Front-End with Elastic Load Balancer

The front-end of MeAndYou is designed to be scalable in Amazon AWS, using an EC2 AMI virtual disk image of an Apache XAMPP-based web server to create an EC2 Launch Template, that is used by an auto-scaling group, which will automatically create new instances of the web server when the demand for the web service is higher than normal, and terminate those instances about 15 minutes after they are no longer needed. The auto-scaling group will have triggers, that based on a certain level of traffic to any of the web servers, new instances will be generated by AWS automatically. Additionally, all the web server instances, whether they are launched manually in the Console, or via the command line using the AWS Command Line Interface (CLI), or automatically via the auto-scaling group, will all be connected to the Internet domain name of the web service with an AWS Elastic Load Balancer.

To have the web service code on the web server be scalable, it will be necessary to combine common code into common libraries within the website's code repository. This will include layering the PHP code, to separate the different layers of code, such as the template code and markup, business logic code, and database accessibility code.

For this scalable web service experiment, an Access Control List (ACL) will be used to control what traffic is allowed in and out of the AWS Security Group to which the EC2 instances and other resources will be assigned to. The ACL will keep unwanted traffic from viewing or accessing the web service while it is being tested. For this experiment, only traffic from UNH Manchester students and faculty will be allowed to access the web service. A diagram of the front-end can be seen below in **Figure 7**.

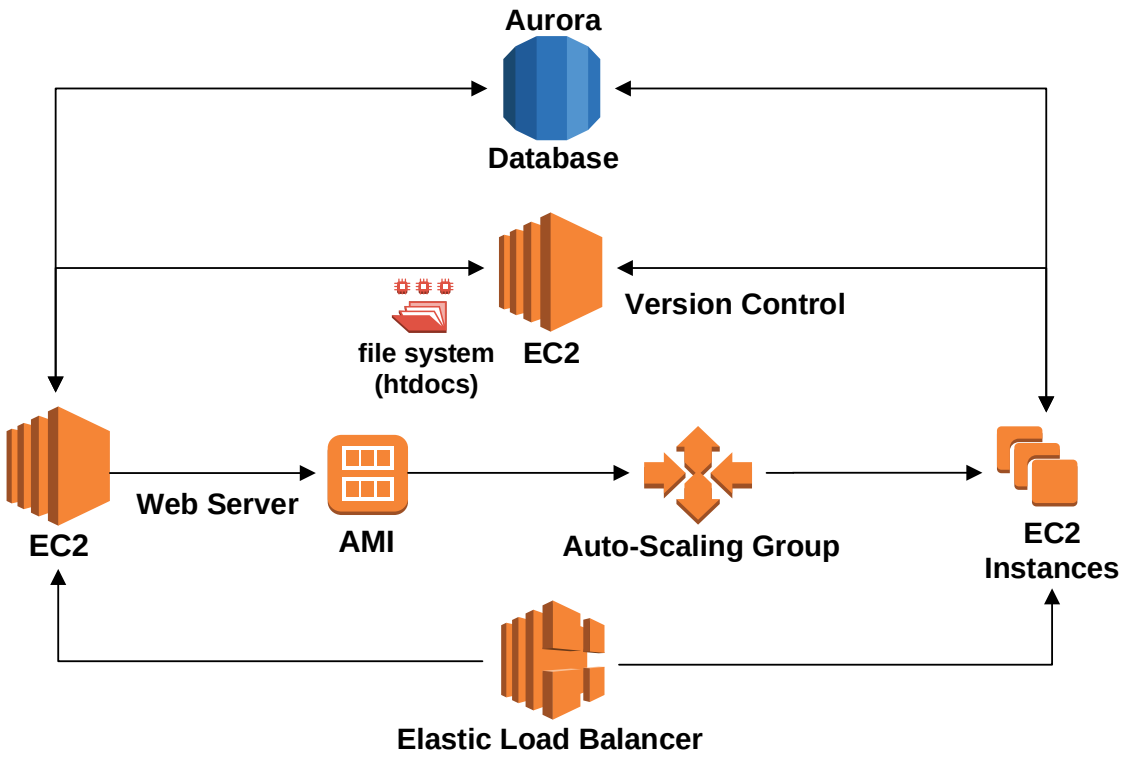

Figure 7: MeAndYou AWS Front-End Diagram

3.3.2 Version Control Instance

An EC2 instance will also need to be setup for Version Control, or source control, of which updates to the web service's code will be made and will be propagated to the web servers. Because there are so many web servers that will be running, this is necessary, to ensure that the updates are applied to the web servers when updates are uploaded via FTP to the Version Control

instance. Without the Version Control instance, it would be necessary to upload the changes to each of the web servers, each time changes are made. If there were 5 web servers at any given time, the changes would have to be uploaded to each of the web servers, and because of the auto-scaling group – it becomes imperative to have the Version Control instance.

3.3.3 Aurora Database Instances

The database for the web service will be hosted in Amazon's Relational Database Service (RDS) under the AWS Aurora system, which supports native MySQL database types. The instance size will be 1TB, which is comprised of 6 EC2 instances – while up to 64TB is supported per set of instances [33]. Each of the web servers will access the database instance to perform Create, Retrieve, Update, and Delete methods (CRUD) on data sets pertaining to the web service, so that information can be shared between multiple users on the site. The database will be scalable, so that if the data becomes larger than the original instance size, it can be migrated in the same datacenter to another, larger database instance.

3.3.4 Match Engine Auto-Scaling Group

The web service works because of a match engine that will run on a separate set of EC2 instances, also in an auto-scaling group, that will all connect to a message queue that receives messages from the front-end each time a new search is created, to alert one of the engine instances to look for a matching record in the searches database table. If no match is found, the Match Engine instance will store the information for use later by the same system. The Match Engine instances connect to the same database instance as the web servers. The match engine scaling can be seen below, in **Figure 8**:

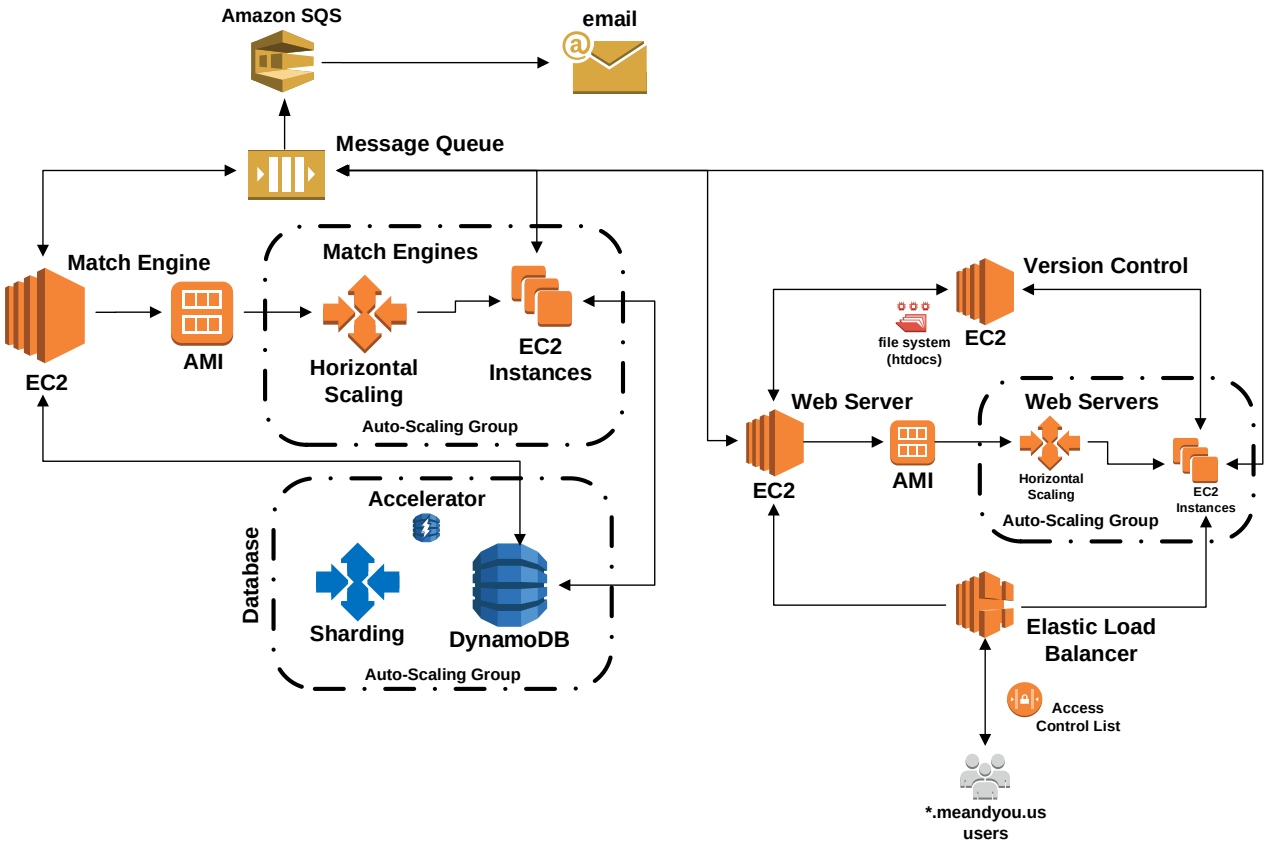

Figure 8: MeAndYou AWS Infrastructure Diagram

3.3.5 Domain Name System Services

This scalable web service will use Route 53 within AWS for DNS, including the assignment of www.meandyou.website and meandyou.website to the load balancer. Other possible domain names will include friendly aliases for the Version Control instance, the original web server instance in which updated versions of the AMI for the GUI will be created, and an alias for the database instance. Since some of these resources could change IP addresses as they are deactivated and reactivated, it may be necessary to use their AWS-issued hostname. For AWS to be able to host the domain name meandyou.website, it will be necessary to assign the domain name servers at MyDomain, where the domain name was registered in the first place.

CHAPTER 4

PROTOTYPE OF MEANDYOU

4.1 Front-End Design

The implementation of the prototype of MeAndYou started by consolidating common code into common PHP code files. The front-end works through an implementation of Apache XAMPP on multiple EC2 instances, one of which was created to make an AMI image in Amazon AWS, and others that were created by cloning the AMI virtual machine image.

The auto-scaling group is made up of additional EC2 instances that duplicate the AMI. Since each instance is a clone of the original MeAndYou-GUI-1 instance, it works essentially in the same way as the original instance, but they all have separate IP addresses, which had to be provisioned and added to the security group that allows traffic to reach the database and the version control system. Since the IP addresses are taken from a pool of available IP addresses, they aren't necessarily right next to each other in a /29 subnet mask like they ideally should be – they are from much larger subnets, because the addresses are made available as other instances within the AWS system are no longer used from the pool of all AWS's IP addresses.

One issue with the front-end instances, was that after RDP started working on MeAndYou-GUI-1, it no longer worked on the duplicated AMI instances for some reason. Even when tightvncserver, xrdp, and other software was re-installed, and settings were reconfigured, RDP was still not working. There was a setting that if enabled – that while creating the AMI, the system

would not reboot, which could cause corruption – but that setting wasn't enabled. Other than RDP not working, everything worked normally on the cloned systems.

A JavaScript authentication script was coded and implemented on the registration form, that checks to see if a username is already taken in the database, since submitting a form for a new user with a username that is already taken would produce a backend error. It uses a setup like Asynchronous JavaScript and XML (AJAX), that has a PHP script that returns 1 if the username is already in use, or 0 if the username isn't in use. The JavaScript app then displays a message, whether it be that the Username is Available, or that the Username is Taken, and if it is taken, it also temporarily disables the registration submit button, so the user cannot submit the form while using a username that is already taken. This of course creates additional traffic on the web server, however it is of the convenience of the visitor registering to use the site to know that they can select that specific username that is available or know otherwise.

JavaScript filtering for the display of searches on the front page of the Dashboard was also added. The issue arose that there was a matched search that wasn't allowing submission of anymore Long Lost Love searches, because of the limit of one per user, so the capability to show all searches was added (even those that are matched) on the front page of the Dashboard, and a drop-down menu was added that allows the user to filter the searches by either search type and/or search status. So, for an instance, the user can select to only display searches that are for "Friend" and are search status "Processing", or "Matched", or "Incomplete", or "Complete". The script also allows the filtering of all searches by the search types of "Crush", "Family", "CoWorker", or "Love" as well. A button was also added for deleting all searches and matches.

Filtering through JavaScript was also added on the notifications page, so that the user can select whether they want to display only new notifications, or only deleted notifications, or all notifications – new and deleted. A button was also added to delete all new notifications.

Several new fields were added to the front-end, including Maiden Name, Emails #2 through #9, Phone Numbers #2 through #5, Addresses #2 through #5, Company Name, and Job Title. A Starting Date and Ending Date field for all address and phone number fields were also added.

The input/output templating system for account.php and accountedit.php, which handles the user's changes to their account profile, as well as search.php and searchedit.php, which handles modifications to searches, all use common code – each for each set of similar pages. The account.php and accountedit.php use a helper page named requirefields.php, which originally started as a function that checked for which required fields are necessary for a specific type of search – but now does other things, such as building the code for a state of domicile selection drop-down menu, automatically, so that the duplicate code can be condensed into one location for the site – at least for those pages. Also, search.php and searchedit.php use a page with helper functions named searchdata.php, which is like requiredfields.php, in the sense that it removes markup generation and Input/Output (I/O) access from the template page.

4.1.1 Layering of Hypertext Preprocessor Code

The large task of refactoring all the common code and common calls in controller.php, model.php, and util.php to be more layered in the sense of what each of the layers of code does has been a major issue – to separate out code that interacts with the database, from business

logic, from templating and markup code that displays and collects information from the user. This layering of MeAndYou's code base is shown on page 40 in **Figure 9**. A similar process of separating code layers was described in "Building Scalable Websites" by Cal Henderson [32].

The process of layering the code included setting up a set of global variables that are used to configure the database with the front-end code, including:

- **$dbms_hostname** – the hostname needed to connect to the database server
- **$dbms_username** – the username needed to connect to the database server
- **$dbms_password** – the password needed to connect to the database server
- **$dbms_database** – the database in the database server that the front-end code should connect to with these credentials.
- **$main_time_offset** – the number of seconds to adjust the time of system messages displayed on the front-end

These global variables make it a lot easier to make changes to configuration, but ideally in the future there should be a configuration server that sends updates of configuration to all the applicable servers. This would allow the instance of topology or interface changes to be propagated to the web servers immediately, rather than waiting minutes for Version Control to synchronize.

Another modification that has been made to the system has been simplifying the way the code works, through common classes and interfaces. Common database calls, such as getting the person.id for a username (e-mail address), are all in util.php. Each of those functions uses util::connect(), as well as util::disconnect() – which disconnects the session from the database. This allows transactions with the database to be quick and asynchronous.

MeAndYou Codebase Architecture

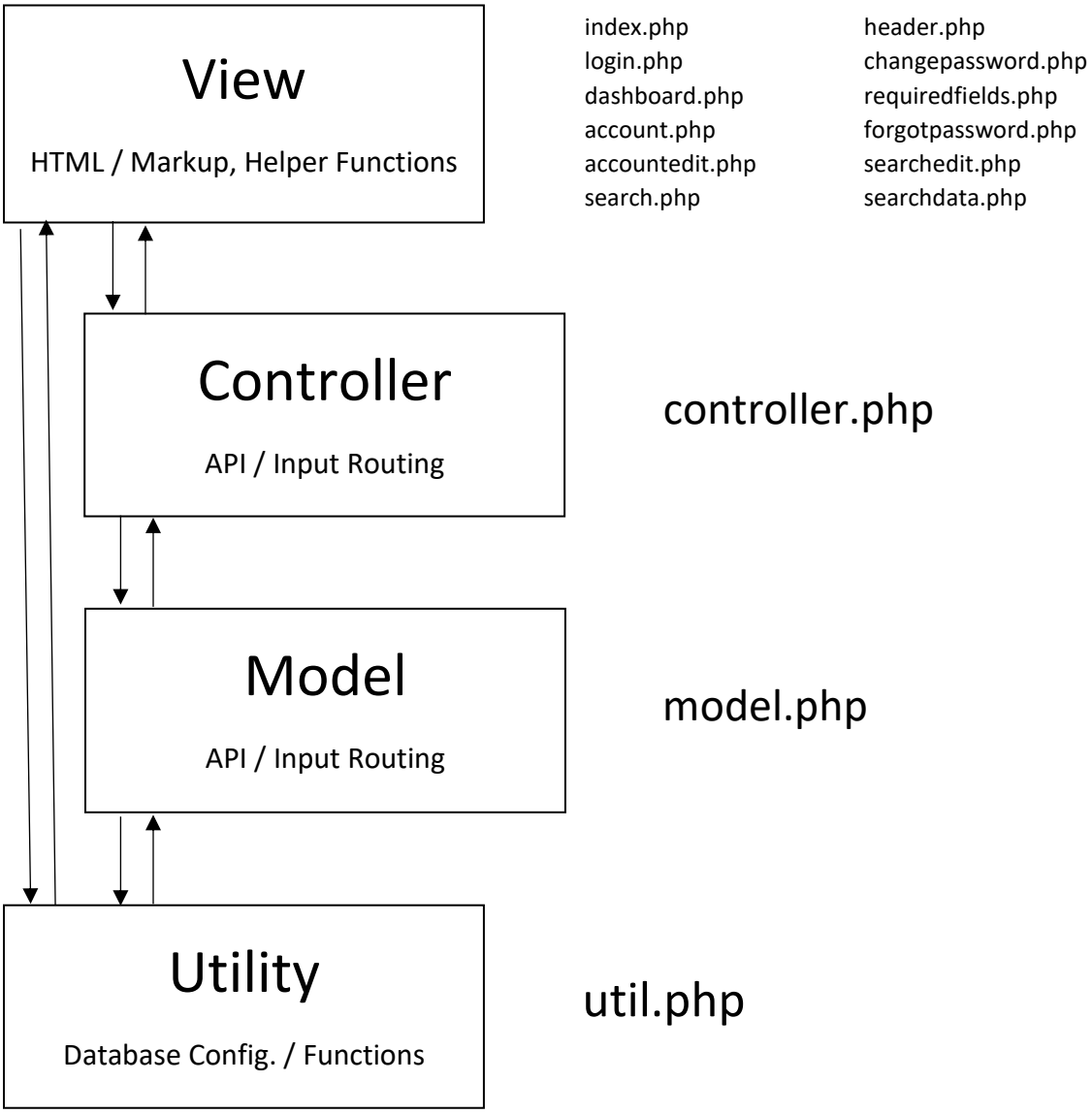

Figure 9: MeAndYou Codebase Architecture

The database connections are all closed after they have completed their transactions, so that other connections can connect – and there is no one over-arching process that ties up the database, which allows different connections for different types of processes that access the

database. These functions can be accessed by the front-end by including util.php, however it is usually model.php that transacts with util.php, and controller.php that transacts with model.php.

The controller.php file doesn't include its' functions in a class, like model.php and util.php do. This allows the controller.php's functions to be called without referring to the class name first, like model:: or util::, so the controller.php functions are considered global functions with respect to the front-end markup code. In controller.php, the code looks for specific strings that are posted in the $_SESSION array, or $_POST array, such as $POST['login-submit'], which tells the program to process the login function. This allows controller.php's code to know what environmental variables are being submitted from the forms from the markup and templating layer, based on looking for one specific name of an element, such as a Submit button. When the Submit button is clicked, the form data from that page is sent to controller.php, which determines which form was submitted, and what it should do with the information. It packages the data from the form into an $array variable, that is then passed to model.php through a function call. Previously, the variables were hard coded into the function calls, so one function call may have had 15 different parametric variables. Simplifying the input with an array with common names as indexes makes it easier to add additional data elements, or input fields, to any page on the website. The process can almost be automated – and may eventually be in some future version of MeAndYou.

The model.php file includes all its functions in the model class, so they must be called by referring to the class name first. An example of a call to a model.php function is model::registerNewUser(), which has a single parametric variable named $array – which is just an array of passed values from the newUserRegistration() function in controller.php. It then

makes decisions based on business logic, such as “Should I allow this new user to register?” by asking questions such as “Does this user account already exist?”. It now includes more security on the backend connected with the front-end, so that rather than displaying an error that states that the user account exists after the form was submitted, it asks the user to enter another e-mail address for their account username – and then determine if it’s the same user as another login with JavaScript. In the future, it could determine if the user is trying to hack into another user’s account by checking to see if the account exists, and then ban that visitor from logging in for a specified amount of time.

In util.php, there are some common functions, such as util::queryCountRows(), which takes in a single parametric variable of a $query_string, which allows the calling function to know how many rows there are in the database, based on the $query_string. An example use of this function, is util::numUsers(), which takes in no parametric variable, and simply returns the number of registered users according to the database. It does this by passing the $query_string value of "SELECT id FROM " . $dbms_database . ".person;" to util::queryCountRows(). This simply means, “Count the number of rows in the person table, in the $dbms_database database”.

If the web service is to be scalable, it must have components that are decoupled that can withstand multiple teams working on improving the web service as it grows from a single-person operation, to a team of programmers, and then possibly to a team of small groups of programmers that all work on different layers and components of the web service at a time. The system must be capable of recovering from errors in programming or systems malfunctions when it is scalable to hundreds of millions of users.

4.1.2 Amazon Machine Image with Auto-Scaling Group

An auto-scaling group was initiated by first creating a launch template inside the AWS Console, where the AMI to use for launching instances was specified. The AMI is a virtual disk image taken of a virtual machine that is running on an EC2 instance. In the launch template, how much disk space that was needed, in addition to the optional AMI volume (none), and what type of EC2 virtual hardware including processing power and Random Access Memory (RAM) that was needed was also specified.

Once the launch template has been created, the auto-scaling group was setup, which was defined as having a maximum of 5 instances total, with a minimum of 1 instance running at any time. The original instance (MeAndYou-GUI-1) was kept running on the load balancer as well, and the auto-scaling group was added to the load balancer, so that now there are at least two instances serving the website – with up to 6 instances if it gets busy. Adding more instances is just one minor change in the auto-scaling group configuration – it could easily be increased to 100,000 instances. AWS does the rest.

It's nice to be able to select the hardware specs needed for each subsequent EC2 instance in the launch template – it's not locked down to the original specifications of the AMI. Originally, a t2.large instance was used, however for the launch template, a t2.medium instance size was used instead, because they would cost less money. The t2.large has 2 virtual CPUs, just like the t2.medium, however the t2.large has 36 CPU credits per hour, and 8GB of memory, compared to just 24 CPU credits per hour and 4GB of memory for the t2.medium. It's important to choose the

right size of instance for the application that is being run [31]. What would be more ideal, would be an EC2 instance that scales between a range of resources.

4.2 Version Control

A Version Control system that uses an EC2 instance, loaded with Apache XAMPP (which runs ProFTPd) allows the developers of MeAndYou to upload new files to the FTP site, and a modification to the MeAndYou-GUI-1 instance was created that added the following command to run every 2 minutes on the machine by adding it to the system's crontab:

```
*/2 * * * * rsync -avz --rsh="ssh -i /home/ubuntu/.ssh/id_rsa"
ubuntu@versioncontrol.meandyou.website:/opt/lampp/htdocs /opt/lampp/
```

whereas versioncontrol.meandyou.website points to the IP address of the Version Control instance – and the rsync command tells the Linux operating system to grab all the files located in the folder /opt/lamp/htdocs/ on the version control system and place the directory structure of them in /opt/lamp/ on the host machine, using rsync. This works because rsync essentially synchronizes the differences in the files, rather than overwriting all of them, so it's quite efficient. The AMI of the GUI instance was also updated, so that the rsync command was included in the instance image, and the MeAndYou-GUI-3 instances were re-launched in the auto-scaling group.

This system was setup for the auto-scaling group, because to make changes to the website's source code and static files, there needed to be a central store for the files – and GitHub wasn't implemented on the web servers, because it would require a password to be entered to access the repository, since the repository isn't public. So instead, one instance was created that would then prevent from having to FTP to each front-end instance every time a change to one

file needed to be made, to upload the file to every front-end instance – not to mention that the number of front-end instances could change over time based on how much load is on the web service. Otherwise, the code updater would have to go and look in the AWS Console to figure out what the IP addresses of the GUI instances were, and this also would not have solved the issue of when an instance first creates its self from the auto-scaling group. The solution with the Version Control instance solves all these problems, and because the rsync is synchronizing the files, it doesn't use up any additional bandwidth that isn't necessary.

It's important to note that the public Rivest-Shamir-Adleman (RSA) cryptosystem key had to be accepted on the GUI instances, first on the AMI copy of the instance, and then it was replicated to the other auto-scaling group instances. Without accepting the public RSA key, rsync would not be working.

The design for the version control system could be improved, should there be hundreds of GUI instances because it would need to be so the instances only initiate rsync when there has been a change to the Version Control system. An idea that there could be a PHP file that changes a numerical version number each time a file is uploaded through the web server, so it would automatically increment the version number, and then each time an instance goes to check to see if there are any updates, it would first query the web server on the Version Control instance to determine if the version number is higher than the last version that the script downloaded – and if so, then run rsync. This would still require the crontab job, however it wouldn't require that the GUI instances SSH into the version control system every 2 minutes to look for updates even if there are no updates yet.

4.2.1 GitHub Repositories

Several GitHub repositories were setup for the project, including one for the front-end, one for the database, and one for the match engine. They may be relinquished to future classes at UNH for further development of MeAndYou. The repositories are as follows:

- **Front-End:** https://github.com/mce123/MeAndYou-www.git
- **Database:** https://github.com/mce123/MeAndYou-Database.git
- **Engine:** https://github.com/mce123/MeAndYou-Engine.git

Obviously, they can't be made public because there are histories of user credentials in the source code that would still be there in the history, even if the user credentials were removed. There's also personal information in the Database files from previous classes, that may eventually need to be encrypted for privacy purposes.

4.3 Database

The database for MeAndYou is a MySQL-based database, hosted in Amazon AWS under the Relational Database Services (RDS). Specifically, it is using Aurora, which uses six EC2 instances per database, and it can handle up to 64TB of data per database instance [33]. MeAndYou uses a relational database scheme, with around a dozen tables for various purposes. The person table contains some of the user profile settings, while others are stored in the attributes table. The attributes table stores attributes that relate to either a person record, or a search record, and the software of MeAndYou determines which records go to which records with respect to persons and searches. Since the person and search attributes are stored in the same table, it will get rather large over time with hundreds of millions of users. Each user would take up a minimum of 12 records in the attributes table, and each search takes up about the

same – whereas each record is a row in the attributes table. This means that for 100 million users, there could be as many as 1 billion records in the attributes table, because of the number of searches that each user could take up, in addition to their own attributes for their person record.

With the size of the attribute table in mind, it would be better to switch the entire MeAndYou database to DynamoDB in the future, as that can handle a lot more records than MySQL-based Aurora. The attribute table could be split up into two separate NoSQL databases, one being the person-related records, and the other being the search-related records.

The way that the database is designed complements the front-end, such as using ENUM()'s to limit what type of input can go into specific fields in the database. For an example, in the searches table, there is an ENUM('Crush', 'Friend', 'Family', 'Love', 'CoWorker'), which limits the input to those values – especially because it wouldn't be good for someone to change one piece of code in the front-end and suddenly it's putting types of searches in the database that the match engine can't handle. Another example of how the database helps the front-end, is with the attributeType table. It has columns called "regex", "placeholder", and "title", which are all front-end fields that go in a form entry element, so that the input can be regulated from the database, as the front-end code just loads the values from the database and puts them into the templates of the front-end's display. A Crow's Feet Diagram of the database model is shown on the next page in **Figure 10**.

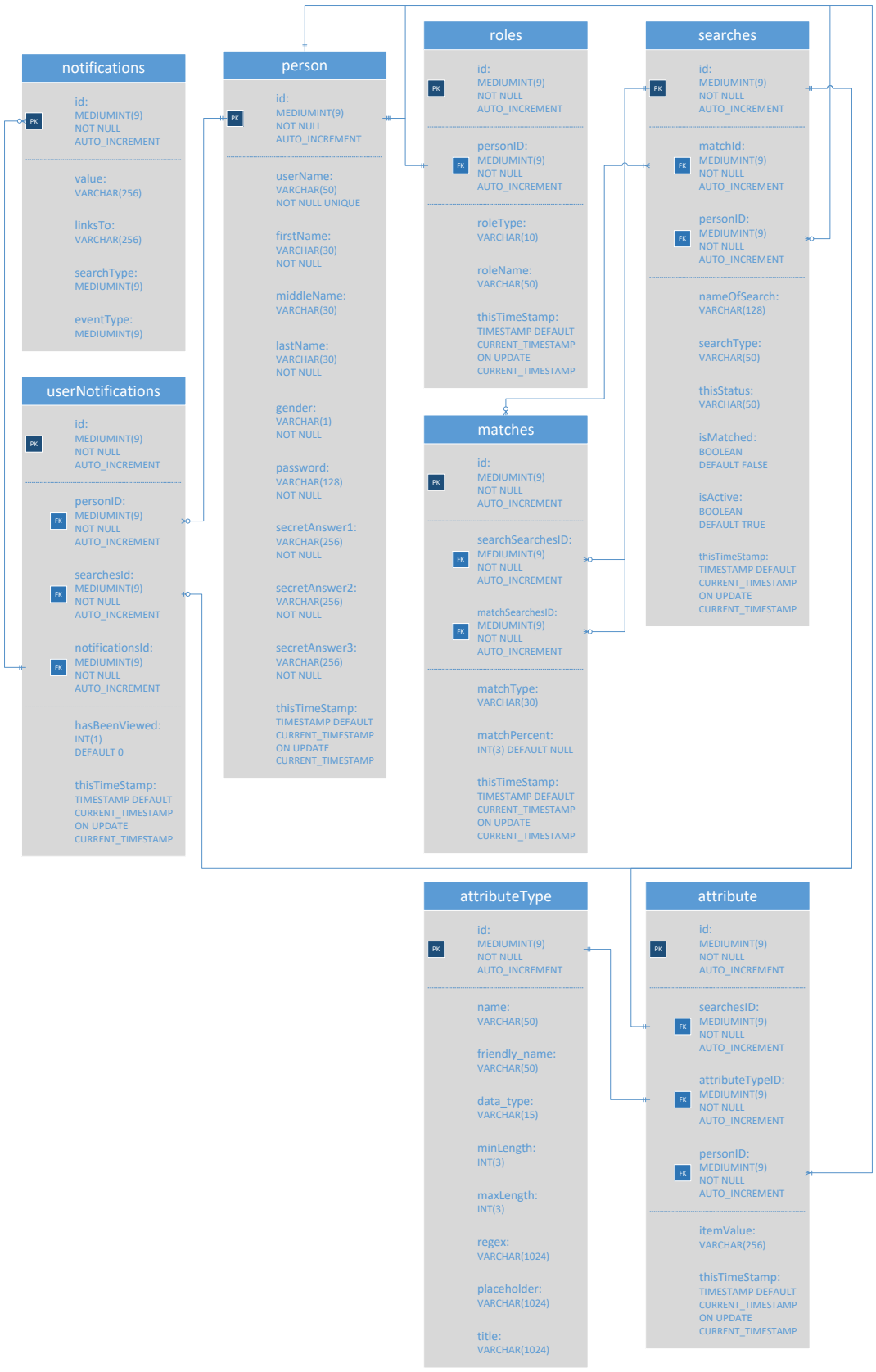

Figure 10: MeAndYou Crow's Feet Database Diagram

4.4 Match Engine

The engine is currently designed as such that it must have all the required input fields in the database filled out, or otherwise it will crash entirely – not just for one record, but for all subsequent records. There was not a lot of error correction / mitigation built into the engine in the first place. As a result, let's say for an example one of the search records doesn't have a birthday in the attributes table, and the birthday field is required – then the engine will crash. The required fields for a valid search are last name, date of birth, gender, and State of domicile. There must also be a legitimate search record created in the searches table, which must include an **id** – which is auto-incremented, a **personId** (based on the person record that created the search), **nameOfSearch**, **searchType**, **thisstatus** – which is usually set to "Incomplete", **isActive** – which is set to 1 by default, and **thisTimeStamp** – which is set to the date timestamp of the system when the search record is created.

The match engine is a separate system in need of its own instance, which currently uses a calculation of the distance between two strings with the Jaro-Winkler distance algorithm. The algorithm returns a value between 0 and 1, depending on how close the two strings match. A returned value of 0 indicates the there are no similarities, while a returned value of 1 indicates an exact match.

The matching agent was also recoded to output the status of what the engine is doing at any moment to the command prompt, and an example of that output is as follows:

```
Calling UpdateNotifications from MatchEngine.cs: 469 false Crush
UpdateNotifications: 469 False Crush
SearchMgr.cs: GetSearchAttributes for 469
```

SearchMgr.cs: GetPeople
Calling UpdateNotifications from MatchEngine.cs: 23 false Crush
UpdateNotifications: 23 False Crush
SearchMgr.cs: GetSearchAttributes for 23
SearchMgr.cs: GetPeople
SearchMgr.cs: GetPerson
SearchMgr.cs: GetSearches

Note that for the match engine, which was designed to conduct stream processing of real-time message ingestion from the GUI, when a search is submitted, to a publish/subscribe system, as evident in **Appendix B: Pseudocode for Improved Match Engine**, there was not enough time to implement this new version. The new engine infrastructure that was designed, would allow an auto-scaling group of Match Engine instances to process the data, one message at a time, potentially from hundreds of millions of concurrent users. This would allow only one engine instance to operate on the new search message, and it would allow that node to then search through the entire database to see if there were any matching records that would make it a match. If it finds any records that would make a match, then it sets those records to matched and creates the match record in the database, and updates all the notifications that apply to those searches. However, if the engine instance doesn't find a match in the database, it simply stores the search record back into the database, so that perhaps at some point in the future it can be matched by another search, or it will be removed if it gets too old.

4.4.1 Implemented Match Engine Re-Coding

The implementation of the match engine was configured / re-coded to support multiple match types. These search types include Crush, Friend, Family, Long-Lost Love, and CoWorker. It

was also ported from a Windows-based executable to a Linux-based executable using MonoDevelop.

4.5 Domain Name System

The Domain Name System (DNS) was configured by setting up the DNS servers in an existing domain name, meandyou.website. The domain name was registered through MyDomain. The following zones in AWS via Route 53 were configured. This caused meandyou.website and www.meandyou.website to redirect to the load balancer. Several sub-domains that helped to connect to instances throughout the project were configured:

- **versioncontrol.** – for redirecting traffic from the GUI nodes using rsync, as well as having the ability to RDP and SSH into the server for various reasons, such as committing and pushing the latest version of the code to GitHub.
- **www1.** – for redirecting traffic to the first version of the GUI instances, so that updates can be made to the original system and then update the Amazon Machine Image (AMI) and reload it into the auto-scaling group.
- **mysql.** – for redirecting traffic to the database, from the GUI nodes, and from the Engine instance(s). Also, for editing the database using MySQL Workbench.

Having the ability to immediately update the values for these subdomains was critical, as there could be changes to IP addresses when nodes are stopped and restarted. One issue that arose was *. and www. were originally forwarded to the IP addresses of the load balancer, and it was not known that the IP addresses would change over time. It had to be forwarded to the hostname of the load balancer instead.

CHAPTER 5

DISCUSSION

5.1 Testing of MeAndYou Prototype Implementation

5.1.1 Graphical User Interface Testing

The only proper way to fully test out the front-end's GUI and the load balancer was to use a program that surfed the website, simulating real user activity for the web services, including placing searches, viewing searches, viewing and modifying a user profile, and deleting searches and matches. This was made possible using Selenium, a web driver used with Python code, to create a script that automatically accessed the web service like a normal user would [34]. The script used threading to open 8 different Firefox windows, that created a simulated user load on the load balancer.

The functionality of the AWS Auto-Scaling Group was tested with this Python script, to ensure that it spins up the extra instances when it gets busy, and terminates them shortly after it is no longer busy. The testing was successful, with the auto-scaling group being configured to activate additional instances when there were more than 100 requests per health check period of 300 seconds for each of the two web server instances that were already running. The success of the auto-scaling group to activate additional instances is shown on the next page in **Figure 11**.

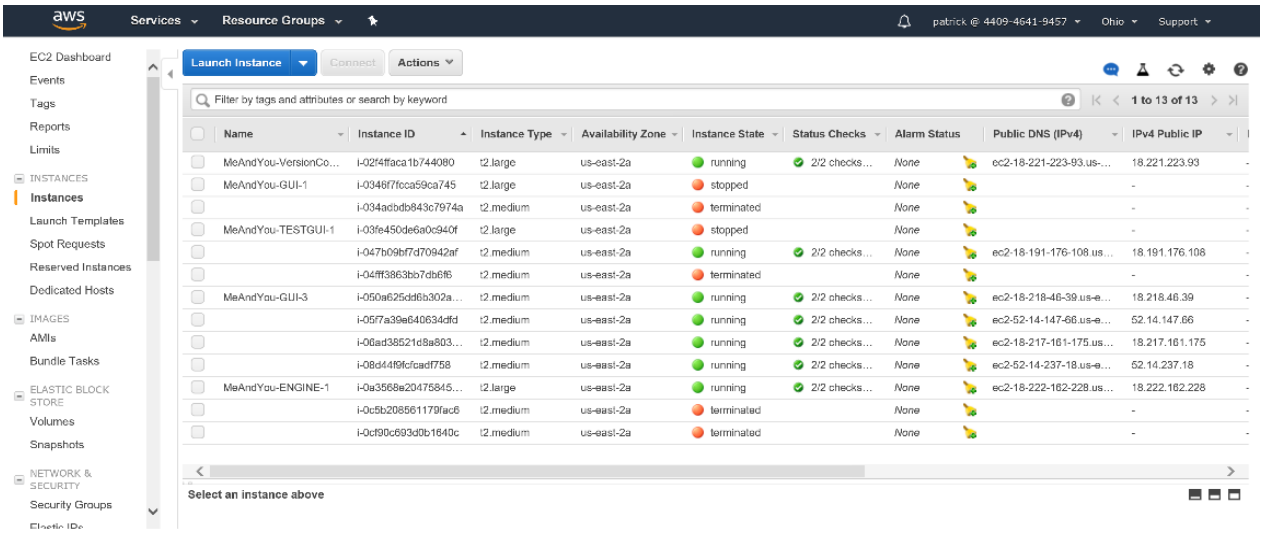

Figure 11: Amazon AWS Console – EC2 Instances Started by Auto-Scaling Group

5.1.2 Match Engine Accuracy Testing

A Microsoft Excel spreadsheet was used to generate sample data, based on a set of tests, and the spreadsheet was exported as Comma Separated Values (CSV) files, and then converted to SQL files, that were imported into the database while the MeAndYou-ENGINE instance was not running, to ensure that the search records would not compute the wrong results while they were being imported, such as the lack of data attached to specific searches. Usually the data is generated based on the search forms on the website, so it's not possible to import the wrong type of data that way, or the lack of certain data, due to security restrictions on the website in the form of registry expression validation, and PHP security in the form of validity checks. Once the data was imported, the MeAndYou-ENGINE instance was restarted so the 400 search records could be processed into up to 200 match records. The 400 search records linked back to 201 person records, since there is one less match record per set of user records if they all point to

one record, since the record can't search for its self. It takes 400 search records, because the system needs 200 searches on both sides of the search, since it is a black box search method.

Tests that were conducted included providing the incorrect first or last name, providing the incorrect birthdate which was either +1 day, +1 month, or +1 year, or providing the incorrect state of domicile, or providing the wrong email address, and also combinations between these different types of errors – such as wrong email and wrong name(s), wrong email and wrong date of birth, wrong email and wrong state of domicile, or wrong name(s) and wrong date of birth, or wrong name(s) and wrong state of domicile, or wrong date of birth and wrong state of domicile. Of these tests conducted, 25% were matched to Crush type, 25% to Family type, 25% to Friend type, and 25% to Long-Lost Love type. The type of match didn't make any difference in the way that the matches were made or not made.

The engine tests performed resulted in 332 of the 400 searches being matched together, or in other words 166 match records, while another 68 search records failed, or 34 match possibilities that didn't result in a match occurring. Of the potential matches that didn't occur, they pretty much all resulted from issues with either the wrong name being entered, or there not being a first name provided. The same would have occurred had the last name not been provided, while it also may have caused the engine to crash because to requires the last name record to be in the database.

Of the incorrect first or last name on either side of the searches, these included the following possibilities:

- Incorrect first or last name on either or both sides, even with correct e-mail addresses on both sides
- Incorrect first or last name on either or both sides, even when correct birthdate is provided on both sides
- Incorrect first or last name on either or both sides, even when correct state is provided on both sides

These results show how specific the issues are that would prevent a potential match from being matched.

5.2 Conclusion

MeAndYou has now been deployed with a highly scalable model, in which new nodes can be added at the datacenter level and allow the model to scale into the hundreds of millions of users. A lot has been learned about scalable web service development along the way, and it all worked out well – to the point that this could be offered as a service through MCE123, to large companies that need scalable web service infrastructure. Testing of the various software environments was conducted, and the theory worked.

There can never be enough planning on something that scales to hundreds of millions of users. It's going to take a large staff of programmers to build the web service to be competitive in this market, with other companies such as Facebook and Twitter. However, it has been proven that MeAndYou can scale up to that size – and now it's just a matter of whether people want to

use it, and whether a business model to monetize it would support the expenses that it would cost the company to run such a large site for the public.

Someday, hopefully someone will find the interest in revamping this project, and possibly expose it to the UNH student body. Only then can it truly be tested for practicality. It will require someone that can implement the publish/subscribe message queue with the match engine, which will allow the Match Engine to scale to hundreds of millions of users.

It must be remembered that technology-agnostic and brand-agnostic architectures are vital to ensure that the products and services that are combined to provide the hosting services for scalable web service design are the best performers for the cost. While Amazon AWS worked well for this research, it may potentially not be the best solution for your site's architecture. However, regardless of the size or complexity of your project, this sample project serves as testimony that it's possible to scale virtually any website into a valuable web service that customers could use to generate revenue for an online company. Then it's just a matter of marketing the web service to obtain a customer base.

BIBLIOGRAPHY

[1] Wikipedia. 2018. Healthcare.gov From Wikipedia, the free encyclopedia. Retrieved July 12, 2018 from: https://en.wikipedia.org/wiki/HealthCare.gov.

[2] Martin L. Abbott and Michael T. Fisher. 2015. The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise (2nd. ed.). Addison-Wesley Pearson Education, Inc., Old Tappan, NJ.

[3] Wikipedia. 2018. Service-level agreement From Wikipedia, the free encyclopedia. Retrieved June 12, 2018 from: https://en.wikipedia.org/wiki/Service-level_agreement.

[4] km. 2018. Uptime and downtime with 99.9% SLA. Retrieved July 10, 2018 from: https://uptime.is/.

[5] Jeff Barr. 2017. Amazon Elastic Container Service for Kubernetes. Retrieved July 27, 2018 from: https://aws.amazon.com/blogs/aws/amazon-elastic-container-service-for-kubernetes/.

[6] Microsoft Corporation. 2018. Microsoft Docs: Azure Application Architecture Guide. Retrieved June 12, 2018 from: https://docs.microsoft.com/en-us/azure/architecture/opbuildpdf/guide/toc.pdf?branch=live.

[7] Wikipedia. 2018. Microservices From Wikipedia, the free encyclopedia. Retrieved June 12, 2018 from: https://en.wikipedia.org/wiki/Microservices.

[8] Mike Wasson and Stuart Celarier. 2017. Microsoft Azure: Microservices architecture style. (November 2017). Retrieved June 12, 2018 from: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices.

[9] Margaret Rouse and Ivy Wigmore. 2013. WhatIs.com: Definition: 3Vs (volume, variety and velocity). (February 2013). Retrieved June 30, 2018 from: https://whatis.techtarget.com/definition/3Vs.

[10] Mike Wasson. 2017. Microsoft Azure: Big Data Architecture Style. (November 2017). Retrieved June 16, 2018 from: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/big-data.

[11] Khtan66. 2016. File: Kubernetes.png From Wikipedia Commons, the free media repository. (November 2016). Retrieved June 30, 2018 from: https://commons.wikimedia.org/w/index.php?curid=53571935.

[12] Wikipedia. 2018. Kubernetes From Wikipedia, the free encyclopedia. Retrieved June 30, 2018 from: https://en.wikipedia.org/wiki/Kubernetes.

[13] Kelsey Hightower, Brendan Burns and Joe Beda. 2017. Kubernetes Up & Running: Dive Into The Future Of Infrastructure. O'Reilly Media, Inc., Sebastopol, CA.

[14] Brendan Burns. 2018. Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. O'Reilly Media, Inc., Sebastopol, CA.

[15] Statista. 2018. Statista: The Statistics Portal: Number of monthly active Facebook users worldwide as of 1st quarter 2018 (in millions). Retrieved June 11, 2018 from: https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/.

[16] Statista. (2018). Statista: The Statistics Portal: Leading countries based on number of Facebook users as of April 2018 (in millions). Retrieved June 11, 2018 from: https://www.statista.com/statistics/268136/top-15-countries-based-on-number-of-facebook-users/.

[17] Zephoria Digital Marketing. 2018. Strategic Insights: The Top 15 Valuable Facebook Statistics – Updated May 2018. Retrieved June 11, 2018 from: https://zephoria.com/top-15-valuable-facebook-statistics/.

[18] Facebook. 2018. Facebook For Developers: Graph API: Overview. Retrieved June 11, 2018 from: https://developers.facebook.com/docs/graph-api/overview.

[19] Sean Michael Kerner. 2010. Developer.com: Open Source: Inside Facebook's Open Source Infrastructure. (July 2010). Retrieved June 11, 2018 from: https://www.developer.com/open/article.php/3894566/Inside-Facebooks-Open-Source-Infrastructure.htm.

[20] Steve Campbell. 2010. MUO: How Does Facebook Work? The Nuts and Bolts [Technology Explained]. (February 2010). Retrieved June 12, 2018 from: https://www.makeuseof.com/tag/facebook-work-nuts-bolts-technology-explained/.

[21] The PCMan Website. 2018. The PCMan Website: Your source for fun, free games-web tools-freeware: Free Alexa Page Rank Checker. Retrieved June 11, 2018 from: https://www.thepcmanwebsite.com/alexa_pagerank_checker.php?url=www.facebook.com.

[22] Matt Trifiro. 2016. Data Center Knowledge: Data Center FAQs: The Facebook Data Center FAQ. (September 2010). Retrieved June 11, 2018 from: http://www.datacenterknowledge.com/data-center-faqs/facebook-data-center-faq.

[23] Mike Rogoway. 2015. OREGONLIVE: The Oregonian: Facebook greenlights third Prineville data center: It's Oregon's biggest. (September 2015). Retrieved June 11, 2018 from: http://www.oregonlive.com/silicon-forest/index.ssf/2015/09/facebook_greenlights_third_pri.html.

[24] Facebook for Developers. 2018. Facebook for Developers: Documentation. Retrieved June 12, 2018 from: https://developers.facebook.com/docs.

[25] Pamela Vagata and Kevin Wilfong. 2014. Facebook: Code: Scaling the Facebook data warehouse to 300 PB. (April 2014). Retrieved July 1, 2018 from: https://code.fb.com/core-data/scaling-the-facebook-data-warehouse-to-300-pb/.

[26] Maxim Panchenko. 2018. Facebook: Code: Accelerate large-scale applications with BOLT. Retrieved July 1, 2018 from: https://code.fb.com/data-infrastructure/accelerate-large-scale-applications-with-bolt/.

[27] Wikipedia. 2018. Match.com From Wikipedia, the free encyclopedia. Retrieved July 15, 2018 from: https://en.wikipedia.org/wiki/Match.com.

[28] Robert L. Mitchell. 2009. ComputerWorld: Online dating: The technology behind the attraction. (February 2009). Retrieved July 15, 2018 from: https://www.computerworld.com/article/2530789/networking/online-dating--the-technology-behind-the-attraction.html.

[29] Ravi Kalakota. 2015. PracticalAnalytics.co: Love, Sex and Predictive Analytics: Tinder, Match.com, and OkCupid. (Mary 2015). Retrieved July 15, 2018 from: https://practicalanalytics.co/2015/05/29/love-sex-and-predictive-analytics-tinder-match-com-and-okcupid/.

[30] Mazdak Hashemi. 2016. Twitter: Engineering: Infrastructure: The infrastructure behind Twitter: efficiency and optimization. (August 2016). Retrieved July 15, 2018 from: https://blog.twitter.com/engineering/en_us/topics/infrastructure/2016/the-infrastructure-behind-twitter-efficiency-and-optimization.html.

[31] Amazon AWS. 2018. AWS: Amazon EC2 Instance Types. Retrieved July 1, 2018 from: https://aws.amazon.com/ec2/instance-types/.

[32] Cal Henderson. 2006. Building Scalable Websites: Building, Scaling, and Optimizing the Next Generation of Web Applications. O'Reilly Media, Inc., Sebastopol, CA.

[33] Amazon AWS. 2018. AWS: Amazon Aurora. Retrieved July 14, 2018 from: https://aws.amazon.com/rds/aurora/?nc2=h_m1.

[34] SeleniumHQ. 2018. SeleniumHQ: Browser Automation. Retrieved July 8, 2018 from: https://docs.seleniumhq.org/.

APPENDICIES

APPENDIX A :: MeAndYou Front-End Source Code

controller.php Snippet:

```
session_start();

ini_set('display_errors', 1);
ini_set('display_startup_errors', 1);
error_reporting(E_ALL);

require_once("model.php");
require_once("util.php");

//Calls the Corresponding Function depending upon what Submit Button the User has clicked
if (isset($_POST['login-submit'])) {
        processLogin();
} else if (isset($_POST['registerButton'])) {
        newUserRegistration();
} else if (isset($_POST['CrushSearch'])) {
        addSearch(1);
} else if (isset($_POST['FriendSearch'])) {
        addSearch(2);
} else if (isset($_POST['FamilySearch'])) {
        addSearch(3);
} else if (isset($_POST['LoveSearch'])) {
        addSearch(4);
} else if (isset($_POST['CoWorkerSearch'])) {
        addSearch(5);
} else if (isset($_POST['logout'])) {
        logout();
} else if (isset($_POST['accountEdit'])) {
        accountEdit();
} else if (isset($_POST['forgot-submit'])) {
        forgotPassword();
} else if (isset($_POST['changepassword-submit'])) {
        changePassword();
} else if (isset($_POST['deleteSearch'])) {
        deleteSearch();
} else if (isset($_POST['deleteMatch'])) {
        deleteMatch();
} else if (isset($_POST['deleteNotification'])) {
        deleteNotification();
} else if (isset($_POST['searchEdit'])) {
        editSearch();
```

```
} else if (isset($_POST['updateSearch'])) {
        updateSearch();
}

function newUserRegistration() {
    //Generate Password Hash
    $password           = $_POST['password'];
    $array['hash']      = hash('sha256', $password);
    $array['email']     = $_POST['userName'];
    $array['firstName'] = $_POST['fname'];
    $array['middleName'] = $_POST['mname'];
    $array['lastName']  = $_POST['lname'];
    $array['nickName']  = $_POST['nname'];
    $array['answer1']   = $_POST['answer1'];
    $array['answer2']   = $_POST['answer2'];
    $array['answer3']   = $_POST['answer3'];
    $array['phoneNumber'] = $_POST['phoneNumber'];
    $array['gender']    = $_POST['gender'];
    $array['birthdate'] = $_POST['dob'];
    $array['address']   = $_POST['address'];
    $array['city']      = $_POST['city'];
    $array['state']     = $_POST['state'];
    $array['zipCode']   = $_POST['zip'];
    $_SESSION['username'] = $array['email'];
    $_SESSION['password'] = $password;

    $regcomplete = model::registerNewUser($array);

    if ($regcomplete) {
       $_SESSION['logged_in'] = TRUE;
       echo header("refresh:0;url=dashboard.php");
    } else {
       $_SESSION['logged_in'] = FALSE;
       echo header("refresh:4;url=login.php");
    }
}
```

APPENDIX A :: MeAndYou Front-End Source Code

model.php Snippet:

```
ini_set('display_errors', 1);
ini_set('display_startup_errors', 1);
error_reporting(E_ALL);
require_once("util.php");

global $dbms_database, $CONFIG_attributeType_required;

        class model{

                static function registerNewUser ($array) {
                        global $dbms_database;
                        $mysqli = util::connect();

                        //Check that user account doesn't already exist - if it does, display
                        an error and return 0
                        $accountexists = util::accountExists($array['email']);
                        if ($accountexists > 0) {
                                echo 'Error - Account Already Exists.';
                                return 0;
                        }

                        //INSERT statements don't pertain to a Search, so we set
                        $searchesId to 0
                        $searchesId = 0;

                        //Insert values into person table
                        $query_string = "INSERT INTO " . $dbms_database . ".person
                            (person.userName, person.password, person.firstName,
                            person.middleName, person.lastName, person.gender,
                            person.secretAnswer1, person.secretAnswer2,
                            person.secretAnswer3) VALUES ('" . $array['email'] . "','" .
                            $array['hash'] . "','" . $array['firstName'] . "','" .
                            $array['middleName'] . "','" . $array['lastName'] . "','" .
                            $array['gender'] . "','" . $array['answer1'] . "','" .
                            $array['answer2'] . "','" . $array['answer3'] . "');";
                        $result = mysqli_query($mysqli, $query_string);

                        //Get person.id for $email
                       $personid = util::checkUserId($array['email']);
                       $transactions = array(
```

```
                    1  => $array['birthdate'],
                    2  => $array['gender'],
                    3  => $array['phoneNumber'],
                    4  => $array['address'],
                    5  => $array['city'],
                    6  => $array['state'],
                    7  => $array['zipCode'],
                    8  => $array['email'],
                    9  => $array['nickName'],
                    10 => $array['firstName'],
                    11 => $array['middleName'],
                    12 => $array['lastName']
                );

                foreach ($transactions as $attributeTypeId => $itemValue) {
                    $result = mysqli_query($mysqli, "INSERT INTO " .
                        $dbms_database . ".attribute (attribute.searchesId,
                        attribute.attributeTypeId, attribute.personId,
                        attribute.itemValue) VALUES
                        ('$searchesId','$attributeTypeId','$personid','$itemValue');
                        ");
                }

                 util::disconnect($mysqli);
                 addNotification ('1', $personid);
                 return 1;
            }
}
```

APPENDIX A :: MeAndYou Front-End Source Code

util.php Snippet:

```
$dbms_hostname = 'mysql.meandyou.website';
$dbms_username = '********';
$dbms_password = '********';
$dbms_database = 'meandyou2';
$main_time_offset = -14400;
$visible[0] = 'visibility: hidden; position: absolute;';
$visible[1] = 'visibility: visible; position: relative;';

class util {

    /* connect - log into the MEANDYOU2 MySQL Database                    */
    static function connect() {
        global $dbms_hostname, $dbms_username, $dbms_password,
            $dbms_database;
        $dbms_connection = new mysqli($dbms_hostname,
            $dbms_username, $dbms_password, $dbms_database);

        //Check Connection - If False, Return Error
        if($dbms_connection->connect_error){
            echo "Failed to connect to MYSQL<br>";
            die($dbms_connection->connect_error);
        }

        //Return the Connection If There Wasn't an Error
        return $dbms_connection;
    }

    /* query - query the MySQL Database                                   */
    static function query($mysqli, $query_string) {
        $result = mysqli_query($mysqli, $query_string);
        if (!$result) {
            printf("Error: %s\n", mysqli_error($mysqli));
            exit();
        } else {
            return $result;
        }
    }
```

```
        /* queryCountRows - counts and returns the number of rows in a specific
             query by its query string from the database
                $query_string - specifies the query that will be made to the
                database.                                                    */
        static function queryCountRows($query_string) {
                $mysqli = util::connect();
                $result = util::query($mysqli, $query_string);
                $num_rows = mysqli_num_rows($result);
                $result->close();
                util::disconnect($mysqli);
                return $num_rows;
        }

        /* numUsers - returns the number of users currently registered in the
             database                                                        */
        static function numUsers() {
                global $dbms_database;
                $query_string = "SELECT id FROM " . $dbms_database .
                    ".person;";
                return util::queryCountRows($query_string);
        }

        /* numMatches - returns the number of matches currently made in the
             Database                                                        */
        static function numMatches() {
                global $dbms_database;
                $query_string = "SELECT id FROM " . $dbms_database .
                    ".matches;";
                return util::queryCountRows($query_string);
        }

        /* numMatchesToday - returns the number of matches made today in
             the database                                                    */
        static function numMatchesToday() {
                global $dbms_database;
                $query_string = "SELECT id FROM " . $dbms_database . ".matches
                    WHERE DATEDIFF(NOW(), thisTimeStamp) <= 1;";
                return util::queryCountRows($query_string);
        }
}
```

APPENDIX B :: Pseudocode for Improved Match Engine

Front-End:

- Trigger – New Search Entered at Front-End
- Store database records with attributes and searches entry.
    - Status set to “Incomplete”.
- Send a message to the message queue with information about searches entry.
- Redirect user to a “Processing… We are checking to see if there is a match already in our system. This may take up to one minute to complete.” page, that displays if there was or was not a Match found, not to allow the user to stay on the page for more than 1 minute. Otherwise re-direct the user to a Match Not Found page, with instructions of what to do to get their Match made – i.e. user should contact the other person to make sure they are on the site, and that they are actively searching for them.

Engine:

- Take one message from message queue
- Get search attributes for the searches entry that links to the message from the message queue.
- Search through database of searches records to find matching search.
    - If matching search is found:
        - Set status of searches entry to “Matched”.
        - Create match record.
        - Update notifications – “You have a new Match! Your %SearchType% search for %FirstName% %LastName%”
    - If no match is found:
        - Set status of searches entry to “Processing”.
- Wait 10 seconds and try again.

Cleanup:

- Scan all searches records for old records that are still set to status “Processing”
    - If found old record, delete it, or set isActive to 0.
        - Update notifications
            - “Your search has been removed from MeAndYou because no match was found within the allowed timeframe. You may now re-submit your search to MeAndYou now for an additional period of time.”
    - Update notifications of existing searches.
        - “We still haven’t found a match for your search for %FirstName% %LastName%, however we will keep trying until %ExpirationDate%.”
- Wait 30 minutes and re-run Cleanup Engine.

APPENDIX C :: Python Selenium-based Test Harness Used for Front-End Load Testing

```
#
#   Filename: testwww-v2.py
#   Purpose: Front-End Tests of MeAndYou Website with Threading
#
#   Created By: Patrick R. McElhiney
#   Modified Date: 7/4/2018
#
#
from selenium import webdriver
from selenium.common.exceptions import TimeoutException
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import sys
import os
import argparse
from datetime import datetime
import threading
__author__ = 'MCE123'
verbose = True
num_pages = 0

class bcolors:
    """
    This class defines the blueprint for text decorations for the command prompt.
    Example:  print(bcolors.HEADER + "Hello World!" + bcolors.ENDC)
    Note:    If you use more than one begin text decoration, the ENDC ends all of them. If you only
             want to end only one of the text decorations, you must restart the other ones after
             where you ended them.
    """
    HEADER = '\033[95m'    #Begin Purple Text
    OKBLUE = '\033[94m'    #Begin Blue Text
    OKGREEN = '\033[92m'   #Begin Green Text
    WARNING = '\033[93m'   #Begin Yellow Text
    FAIL = '\033[91m'      #Begin Red Text
    ENDC = '\033[0m'       #End All Text Decorations
    BOLD = '\033[1m'       #Begin Bold Text
    UNDERLINE = '\033[4m'  #Begin Underlined Text
    LINK     = '\x1b'    #Begin or End Hyperlink
    LINK2    = '\e]8;;'

def main():
    """
    This function runs the main program code, and makes calls to subprocedures to process the front-end tests.
    Returns: None
    """
    global verbose
    global num_pages
    default_url = "http://www.meandyou.website/"   #Default URL of Website to test front-end of.
    default_iters = 1          #Default number of iterations of each thread.
    log_file = "tests.log"     #Default .log file to write out results to.
    default_threads = 4        #Default number of threads if not specified.
    logging = False            #Default - do not change this value. Logging will be enabled only if there is write access to the log_file.

    print('\b' + bcolors.BOLD + __file__ + ' by Patrick R. McElhiney, MCE123 (http://www.mce123.com/)' + bcolors.ENDC + '\b')

    #Parse Input Variables
    parser = argparse.ArgumentParser(description='This is a TestWWW script by MCE123.')
    parser.add_argument('-t','--threads', help='Number of Threads to Use', required=False)
    parser.add_argument('-i','--iterations', help='Number of iterations to process per thread.', required=False)
    parser.add_argument('-l','--logfile', help='Absolute or relative path to output logging file.', required=False)
    parser.add_argument('-d','--delete', action='store_false', help='Disable deletion of searches after test iterations.', required=False)
    parser.add_argument('-v','--verbose', action='store_true', help='Toggles Showing of Errors, Other Important Information.', required=False)
    parser.add_argument('-u','--url', help='URL used to access the MeAndYou Website - should be in the format http://www.meandyou.website/', required=False)
    args = parser.parse_args()

    #Update default_threads if input is relevant
    if args.threads != None:
        if int(args.threads) != default_threads and args.threads > 0:
            default_threads = int(args.threads)

    #Update iterations if input is relevant
    if args.iterations != None:
        if int(args.iterations) != default_iters and args.iterations > 0:
            default_iters = int(args.iterations)

    #Update log_file if input is relevant, and enable logging if there is write access to the log_file
    if args.logfile != None:
        if check_file(args.logfile) == False:
            if check_file(args.logfile, 'w') != False:
                log_file = args.logfile
                logging = True
        else:
            log_file = args.logfile
            logging = True
    else:
```

```
        if check_file(log_file) == False:
            if check_file(log_file, 'w') == True:
                logging = True
        else:
            logging = True

    #Update default_url if input is relevant
    if args.url != None:
        if args.url[0:7] != "http://" and args.url[0:8] != "https://":
            args.url = "http://" + args.url
        if args.url[-1] != '/':
            args.url = args.url + '/'
        print(bcolors.WARNING + "The URL you entered was not formatted correctly. I formatted it to:" + bcolors.ENDC)
        print(bcolors.OKBLUE + args.url + bcolors.ENDC)
        change_url = raw_input(bcolors.BOLD + "Does this look correct? Type Y or N: " + bcolors.ENDC)
        while (change_url != "Y" and change_url != "N"):
            change_url = raw_input(bcolors.BOLD + "Does this look correct? Type Y or N: " + bcolors.ENDC)
        if change_url == "Y":
            default_url = args.url
        else:
            print(bcolors.FAIL + bcolors.BOLD + "Please re-run the command with the correct URL argument, in the format: http://www.meandyou.website/" + bcolors.ENDC)
            return None

    if args.verbose != None:
        verbose = args.verbose

    #Initialize blank set of logging messages
    log_messages = []

    #Initialize Start Time
    start_time = datetime.now()
    log_messages.append('New Test Started at ' + str(start_time))
    log_messages.append('Using {} Threads, with {} Iterations Each'.format(default_threads, default_iters))
    vmessage('Using {} Threads, with {} Iterations Each'.format(default_threads, default_iters), 1, 1)

    #Process Request to run test_www for default_iters number of times.
    threader(default_url, default_iters, args.delete, default_threads)

    #Calculate Time Elapsed
    time_elapsed = datetime.now() - start_time

    #Print Time Elapsed
    vmessage('Time elapsed (hh:mm:ss.ms) {}'.format(time_elapsed), 1)
    log_messages.append('Time elapsed (hh:mm:ss.ms) {}'.format(time_elapsed))

    #Print Statistics
    vmessage('Processed {} WWW Pages Located at {}'.format(num_pages, default_url), 0, 1)
    log_messages.append('Processed {} WWW Pages Located at {}'.format(num_pages, default_url))

    #Write out logging
    if logging == True:
        vmessage('Writing out log file to ' + log_file, 1)
        logger(log_file, log_messages)

    vmessage("All Done!", 1, 1)


def cleanup(browser, thread_id):
    """
    This function cleans up all of the searches that are in the MeAndYou website, in case there were some left over from before.
    """
    try:
        for i in range(10000):
            #Go To Dashboard and Delete Search
            browser.get('http://www.meandyou.website/dashboard.php')
            browser.find_element_by_id("deleteSearch").click()
            vmessage("Deleted Search #" + str(i+1) + " for Thread #" + str(thread_id), 1)
    except:
        vmessage("Done Deleting Searches for Thread #" + str(thread_id), 1)

def threader(default_url, num_iters=1, delete_searches=True, max_threads=1):
    """
    Main threader function that creates the threads for the GUI tests.
    """
    threads = []
    for threadID in range(max_threads):
        name = "Thread #" + str(threadID)
        this_thread = createThread(threadID, name, default_url, num_iters, delete_searches)
        wait_timer = 0
        while threading.activeCount() > max_threads:
            wait_timer+=1
        this_thread.start()
        threads.append(this_thread)
    for t in threads:
        t.join()
    vmessage("Exiting Main Thread...", 1)
```

```
class createThread(threading.Thread):
    """
    This class defines the uploadThread blueprint, of type threading.Thread, which calls do_upload
    to upload one file per thread.
    To Call: uploadThread(threadID, name, full_path, filename, bucket_path, count_files)
    Whereas: threadId:       type int, is the ID of the thread assigned by the calling function
             name:           type str, is the name of the thread assigned by the calling function
             default_url:    type str, is the base URL of the MeAndYou website to test
             num_iters:      type int, is the number of iterations to conduct the user testing for, in each thread
             delete_searches: type bool, is True if searches should be deleted, False otherwise
    """
    def __init__(self, threadID, name, default_url, num_iters, delete_searches):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.default_url = default_url
        self.num_iters = num_iters
        self.delete_searches = delete_searches
    def run(self):
        vmessage("Starting New Thread #" + str(self.threadID), 1)
        test_www(self.default_url, self.num_iters, self.delete_searches, self.threadID)
        vmessage("Exiting Thread #" + str(self.threadID), 1)


def check_file(file_path, mode='r'):
    """
    This function checks to see if the file at file_path is valid. This is a simple file system check, and doesn't look at the contents of the file.
    file_path: absolute or relative file path to file to be checked.
    Returns:   True if file is valid, False otherwise.
    """
    try:
        fin = open(file_path, mode)
        fin.close()
        return True
    except OSError:
        return False
    except IOError:
        try:
            fin = open(file_path, 'w')
            fin.close()
            return True
        except:
            return False


def test_www(default_url, num_iters=1, delete_searches=True, thread_id=0):
    """
    This function runs the front-end tests against the MeAndYou Website for num_iters times. It deletes the searches
    that are created through the testing process if delete_searches is True. Also, thread_id is used to display the
    thread number that is being used to execute the current process, if threading is in use, otherwise it is 0.
    Returns:   None
    """
    global num_pages
    browser = webdriver.Firefox()
    for i in range(1):
        #View About Us Page
        browser.get(default_url + 'aboutus.php')
        num_pages += 1

        #View Terms of Service Page
        browser.get(default_url + 'termsofservice.php')
        num_pages += 1

        #View Privacy Policy Page
        browser.get(default_url + 'privacypolicy.php')
        num_pages += 1

        #Login to site
        browser.get(default_url + 'login.php')
        num_pages += 1
        inputElement = browser.find_element_by_id("login")
        inputElement.send_keys("********")
        inputElement = browser.find_element_by_id("password")
        inputElement.send_keys("********")
        browser.find_element_by_id("login-submit").click()
        for i in range(num_iters):
            try:
                #View Notifications
                browser.get(default_url + 'notifications.php')
                num_pages += 1
            except:
                vmessage(bcolors.FAIL + "Thread #" + str(thread_id) + ", Iteration #" + str(i+1) + ": " + bcolors.ENDC + "Error with Accessing User Notifications", 1)
            try:
                #View Account Information
                browser.get(default_url + 'account.php')
                num_pages += 1
            except:
                vmessage(bcolors.FAIL + "Thread #" + str(thread_id) + ", Iteration #" + str(i+1) + ": " + bcolors.ENDC + "Error with Accessing User Account", 1)
```

```
        try:
            #Update Account Information
            browser.get(default_url + 'accountedit.php')
            num_pages += 1
            browser.find_element_by_id("accountEdit").click()
        except:
            vmessage(bcolors.FAIL + "Thread #" + str(thread_id) + ", Iteration #" + str(i+1) + ": " + bcolors.ENDC + "Error with Editing User Account", 1)

        elementTypes = [["crush", "Crush"], ["friend", "Friend"], ["family", "Family"], ["coworker", "CoWorker"], ["love", "Love"]]

        for element in elementTypes:
            try:
                #Return to Dashboard and Create New Search
                browser.get(default_url + 'dashboard.php')
                num_pages += 1

                #Start Create Crush Search
                browser.find_element_by_id("create" + element[1]).click()

                #Enter Search Parameters
                browser.find_element_by_id(element[0] + "FirstName").send_keys("Patrick")
                browser.find_element_by_id(element[0] + "MiddleName").send_keys("Russell")
                browser.find_element_by_id(element[0] + "LastName").send_keys("McElhiney")
                if (element[1] == "CoWorker"):
                    browser.find_element_by_id(element[0] + "Company").send_keys("MCE123")
                    browser.find_element_by_id(element[0] + "Title").send_keys("Company Founder")
                browser.find_element_by_id(element[0] + "BirthDate").send_keys("1983-10-18")
                browser.find_element_by_id(element[0] + "Gender").send_keys("Male")
                browser.find_element_by_id(element[0] + "NickName").send_keys("Patty Cake")
                browser.find_element_by_id(element[0] + "Address").send_keys("84 Nuthatch Loop")
                browser.find_element_by_id(element[0] + "City").send_keys("Barrington")
                browser.find_element_by_id(element[0] + "State").send_keys("New Hampshire")
                browser.find_element_by_id(element[0] + "ZipCode").send_keys("03825")
                browser.find_element_by_id(element[0] + "Email").send_keys("patrick@mce123.com")
                browser.find_element_by_id(element[0] + "PhoneNumber").send_keys("603-742-5112")

                #Submit Create Crush Search
                browser.find_element_by_id(element[1] + "Search").click()

                try:
                    if (delete_searches == True):
                        #Go To Dashboard and Delete Search
                        browser.get(default_url + 'dashboard.php')
                        num_pages += 1
                        browser.find_element_by_id("deleteSearch").click()
                except:
                    vmessage("Error with Delete Search for " + element[1])
            except:
                vmessage(bcolors.FAIL + "Thread #" + str(thread_id) + ", Iteration #" + str(i+1) + ": " + bcolors.ENDC + "Error with Create " + element[1] + " Search", 1)
        try:
            #Return to Dashboard and Edit Existing Search
            browser.get(default_url + 'dashboard.php')
            num_pages += 1
            browser.find_element_by_id("searchEdit").click()

            #Update Existing Search
            browser.get(default_url + 'searchedit.php')
            num_pages += 1
            browser.find_element_by_id("updateSearch").click()

            try:
                if (delete_searches == True):
                    #Go To Dashboard and Delete Search
                    browser.get(default_url + 'dashboard.php')
                    num_pages += 1
                    browser.find_element_by_id("deleteSearch").click()
            except:
                vmessage(bcolors.FAIL + "Thread #" + str(thread_id) + ", Iteration #" + str(i+1) + ": " + bcolors.ENDC + "Error with Delete Search - All Searches Already Deleted!")
        except:
            vmessage(bcolors.FAIL + "Thread #" + str(thread_id) + ", Iteration #" + str(i+1) + ": " + bcolors.ENDC + "Error with Edit/Delete Search - No Searches Remaining!")

    if (delete_searches == True):
        cleanup(browser, thread_id)

    browser.close()


def logger(file_path, message_list):
    """
    This function writes out iters of message_list, each on a new line, to file_path. If file_path already exists, the function appends the new message_list
    to the bottom of the file.
    file_path:    type str, absolute or relative path to logging file
    message_list:  type list, of messages (type str), each of which will be separated by a newline character when written out to file_path
    Returns:      None
    """
    old_lines = []
    num_old_lines = 0
    fin = open(file_path, 'r')
```

```
    for line in fin.readlines():
        old_lines.append(line.rstrip())
        num_old_lines+=1
    fin.close()
    fout = open(file_path, 'w')
    if num_old_lines > 0:
        for old_line in old_lines:
            fout.write(old_line + '\n')
        fout.write('\n\n')
    for new_line in message_list:
        fout.write(new_line + '\n')
    fout.close()
    return None


def vmessage(message, top_spaces=0, bottom_spaces=0):
    """
    This function displays messages if the verbose setting is set, either by default or via -v, or --verbose.
    Also displays x number of top_spaces above the message, and y number of bottom_spaces below the message.
    """
    global verbose
    if verbose == True:
        for i in range(top_spaces):
            print(" ")
        print(message)
        for i in range(bottom_spaces):
            print(" ")

if __name__ == "__main__":
    main()
```